

DISSERTATION

for the Degree of
Doctor of Natural Sciences (Dr. rer. nat.)

Applying Model-Driven Engineering to Development Scenarios for Web Content Management System Extensions

Submitted by

Dennis Priefer

born February 24, 1987 in Wetzlar.

Department of Mathematics and Computer Science
Philipps-Universität Marburg
University Reference Number: 1180
Marburg, 2021.

Submitted: 10 March 2021
Defended: 2 July 2021

Supervisors:

Prof. Dr. Gabriele Taentzer	<i>Philipps-Universität Marburg</i>
Prof. Dr. Peter Kneisel	<i>Technische Hochschule Mittelhessen</i>

Co-Advisor:

Dr. Daniel Strüber	<i>Radboud Universiteit Nijmegen</i>
--------------------	--------------------------------------

Referees:

Prof. Dr. Gabriele Taentzer	<i>Philipps-Universität Marburg</i>
Prof. Dr. Manuel Wimmer	<i>Johannes Kepler Universität Linz</i>

Philipps



Universität
Marburg

Priefer, Dennis.

*Applying Model-Driven Engineering to Development Scenarios
for Web Content Management System Extensions.*

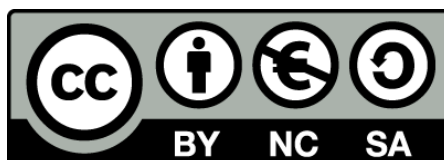
Dissertation, Philipps-Universität Marburg (1180), 2021.

Curriculum vitae

2012, Informatik M.Sc., Technische Hochschule Mittelhessen

2009, Informatik B.Sc., Fachhochschule Gießen-Friedberg

Originaldokument gespeichert auf dem Publikationsserver der
Philipps-Universität Marburg
<http://archiv.ub.uni-marburg.de>



Dieses Werk bzw. Inhalt steht unter einer
Creative Commons
Namensnennung
Keine kommerzielle Nutzung
Weitergabe unter gleichen Bedingungen
3.0 Deutschland Lizenz.

Die vollständige Lizenz finden Sie unter:
<http://creativecommons.org/licenses/by-nc-sa/3.0/de/>

*Meiner Frau Christina und
meinen Kindern Noah und Mila*

Acknowledgements

Now that I have completed this work, I often think of the many people who inspired and motivated me during its creation and who made this work possible in the first place. I would like to take this opportunity to thank them for all their support over the years.

First and foremost, I would like to thank Prof. Dr. Gabriele Taentzer. Her support and trust throughout my academic journey have been the cornerstone of my research. Her insightful feedback has supported me in the conception, structuring and implementation of my publications, including this dissertation. Her expertise and assistance over the years have been invaluable. In the same breath, I also want to thank Prof. Dr. Peter Kneisel for his support, trust and the freedom he has given me to conduct my research in a self-determined way. He has been a mentor to me for years and has been instrumental in shaping me as a person.

Special thanks go to Dr. Daniel Strüber, who took on a mentoring role as a supportive co-author of my publications and helped me to significantly improve the quality of my research. His collaboration brought my work to a higher level, while his guidance helped me to steadily improve my scientific skills over the past years.

Moreover, I would also like to thank Prof. Dr. Manuel Wimmer, who agreed to be my second examiner. His suggestions helped me to improve the details of this dissertation. I would also like to thank Prof. Dr. Bernhard Seeger and Prof. Dr. Christoph Bockisch, who immediately agreed to be members of my examination committee.

I would like to express my gratitude to Wolf Rost for his active support in implementing the JooMDD infrastructure presented in this thesis as well as in conducting and evaluating the empirical work presented. He was by my side not only as a co-author, but also as a loyal companion throughout my doctorate. I also would like to acknowledge Dieudonne Timma Meyatchie, who, in addition to his support in the implementation of the JooMDD infrastructure (code generation, extension extraction), was also an important interlocutor in many discussions during the creation of this thesis.

In the context of the JooMDD infrastructure, I would also like to thank Andrej Sajenko and Alexander Heinz (model extraction), Peter Janauschek (model validation), Max Steinwachs (CI/CD), Lukas Kimpel (web editor), Leon Peulings (IDE plugins), Lukas Schmitt (testing and documentation), Kevin Linne and Samuel Schepp (model interpreters), as well as Mehmet Ali Pamukci as initial user of the infrastructure. Without the support of my colleagues of the project, I would only have been able to present a fraction of the MDE infrastructure presented.

Next, I would like to thank the other (former) members of the Institute of Information Science at the Technische Hochschule Mittelhessen who have constructively supported me during my academic journey over the years by providing feedback during rehearsal talks, on draft papers and during academic discussions. Tim Häuser and James Antrim deserve special mention here. The same applies to my colleagues in the MNI department at the Technische Hochschule Mittelhessen, as well as the colleagues and former members of the software engineering group at the Philipps-Universität Marburg - in particular Steffen Vaupel, Kristopher Born, Nebras Nassar and Felix Rieger.

I would like to thank the Joomla community for the constructive feedback regarding our JooMDD infrastructure. Listing all the names would go beyond the scope of this paper, but I would like to express my very special thanks to Roland Dalmulder and Benjamin Trenkle for their guidance and support. Both of them have always motivated and supported me in improving my MDE approach and making it accessible to the community.

Furthermore, I would like to thank all participants in the empirical studies shown in this paper. On the one hand for the time invested, on the other hand for the constructive feedback.

Moreover, I would like to thank the anonymous reviewers of the published papers in the context of this thesis for their constructive critique and suggestions which helped to improve the quality of my research.

Last but not least, I would like to thank my family and friends who have always motivated me from the idea to the completion of this dissertation. Most of all, I would like to thank my lovely wife Christina for her patience and support over the past years.

Web content management systems (WCMSs) such as WordPress, Joomla or Drupal have established themselves as popular platforms for instantiating dynamic web applications. Using a WCMS instance allows developers to add additional functionality by implementing installable extension packages. However, extension developers are challenged by dealing with boilerplate code, dependencies between extensions and frequent architectural changes to the underlying WCMS platform. These challenges occur in frequent development scenarios that include initial development and maintenance of extensions as well as migration of existing extension code to new platforms. A promising approach to overcome these challenges is represented by model-driven engineering (MDE). Adopting MDE as development practice, allows developers to define software features within reusable models which abstract the technical knowledge of the targeted system. Using these models as input for platform-specific code generators enables a rapid transformation to standardized software of high quality. However, MDE has not found adoption during extension development in the WCMS domain, due to missing tool support. The results of empirical studies in different domains demonstrate the benefits of MDE. However, empirical evidence of these benefits in the WCMS domain is currently lacking.

In this work, we present the concepts and design of an MDE infrastructure for the development and maintenance of WCMS extensions. This infrastructure provides a domain-specific modelling language (DSL) for WCMS extensions, as well as corresponding model editors. In addition, the MDE infrastructure facilitates a set of transformation tools to apply forward and reverse engineering steps. This includes a code generator that uses model instances of the introduced DSL, an extension extractor for code extraction of already deployed WCMS extensions, and a model extraction tool for the creation of model instances based on an existing extension package. To ensure adequacy of the provided MDE infrastructure, we follow a structured research methodology. First, we investigate the representativeness of common development scenarios by conducting interviews with industrial practitioners from the WCMS domain. Second, we propose a general solution concept for these scenarios including involved roles, process steps, and MDE infrastructure facilities. Third, we specify functional and non-functional requirements for an adequate MDE infrastructure, including the expectations of domain experts. To show the applicability of these concepts, we introduce JooMDD as infrastructure instantiation for the Joomla WCMS which provides the most sophisticated extension mechanism in the domain.

To gather empirical evidence of the positive impact of MDE during WCMS extension development, we present a mixed-methods empirical investigation with extension developers from the Joomla community. First, we share the method, results and conclusions of a controlled experiment conducted with extension developers from academia and industry. The experiment compares conventional extension development with MDE using the JooMDD infrastructure, focusing on the development of dependent and independent extensions. The results show a clear gain in productivity and quality by using the JooMDD infrastructure. Second, we share the design and observations of a semi-controlled tutorial with four experienced developers who had to apply the JooMDD infrastructure during three scenarios of developing new (both independent and dependent) extensions and of migrating existing ones to a new major platform version. The aim of this study was to obtain direct qualitative feedback about acceptance, usefulness, and open challenges of our MDE approach. Finally, we share lessons learned and discuss the threats to validity of the conducted studies.

Web-Content-Management-Systeme (WCMS) wie WordPress, Joomla oder Drupal haben sich als beliebte Plattformen für die Erstellung dynamischer Webanwendungen etabliert. Ein großer Vorteil von WCMS ist die funktionale Erweiterbarkeit durch installierbare Erweiterungspakete. Entwickler solcher Erweiterungen stehen jedoch vor der Herausforderung, sich mit großen Mengen an Boilerplate-Code, Abhängigkeiten zwischen Erweiterungen sowie Architekturanpassungen an der zugrunde liegenden WCMS-Plattform auseinanderzusetzen. Diese Herausforderungen treten in gängigen Entwicklungsszenarien auf, welche sowohl die Entwicklung und Wartung von neuen Erweiterungspaketen, als auch die Softwaremigration von bestehenden Erweiterungen auf neue WCMS Plattformen umfassen. Einen vielversprechenden Ansatz zur Bewältigung dieser Herausforderungen bietet der Einsatz von Model-Driven Engineering (MDE). MDE als Softwareentwicklungsmethode sieht vor, fachliche Softwareanforderungen innerhalb von wiederverwendbaren Modellen zu beschreiben, mit dem Ziel, technische Details über das Zielsystem weitestgehend zu abstrahieren. Die Verwendung solcher Modelle als Eingabe für plattformspezifische Codegeneratoren ermöglicht eine automatisierte Übersetzung in standardisierte Software von hoher Qualität. Allerdings hat MDE bei der Entwicklung von Erweiterungen in der WCMS-Domäne aufgrund fehlender Werkzeugunterstützung bisher wenig Akzeptanz erreicht. Während Ergebnisse empirischer Studien in anderen Domänen den Nutzen von MDE aufzeigen, fehlen derzeit empirische Belege für etwaige Vorteile im Kontext der Entwicklung von WCMS Erweiterungen.

In dieser Arbeit werden die Konzepte sowie das Design einer MDE-Infrastruktur für die Entwicklung von WCMS-Erweiterungen vorgestellt. Die vorgestellte Infrastruktur umfasst eine domänenspezifische Modellierungssprache (DSL) für WCMS-Erweiterungen, Modelleditoren, sowie Werkzeuge zur Unterstützung von Forward- und Reverse-Engineering-Prozessen, bestehend aus einem Codegenerator, welcher Modellinstanzen der vorgestellten DSL verwendet, einem Werkzeug für die Code-Extraktion aus installierten WCMS-Erweiterungen, sowie einem Werkzeug zur Informationsgewinnung, welches basierend auf bestehenden Erweiterungspaketen Modellinstanzen der vorgestellten DSL generieren kann. Der Entwurf der gezeigten Infrastruktur befolgt folgenden Prozess: Zunächst wird die Repräsentativität gängiger Entwicklungsszenarien untersucht, indem Interviews mit Entwicklern aus der WCMS-Domäne durchgeführt werden. Als nächstes wird ein allgemeines Lösungskonzept für diese Szenarien vorgeschlagen, welches Rollen, Prozessschritte sowie notwendige MDE-Infrastrukturkomponenten umfasst. Daraufhin werden funktionale sowie nicht-funktionale Anforderungen an eine adäquate MDE-Infrastruktur gesammelt. Um die Anwendbarkeit der vorgestellten Konzepte zu zeigen, wird die JooMDD-Infrastruktur vorgestellt, welche die Entwicklung von Erweiterungen für das Joomla WCMS adressiert, welches im Vergleich populärer WCMS den anspruchsvollsten Erweiterungsmechanismus bietet.

Um Belege für den positiven Einfluss von MDE während der Entwicklung von WCMS-Erweiterungen zu sammeln, werden die Ergebnisse empirischer Studien mit Erweiterungs-Entwicklern aus der Joomla-Community vorgestellt. Dabei werden zuerst Methode, Ergebnisse sowie Schlussfolgerungen eines kontrollierten Experiments mit Entwicklern aus dem akademischen sowie industriellen Bereich präsentiert, welches konventionelle Erweiterungs-Entwicklung mit dem MDE Ansatz vergleicht. Die Ergebnisse zeigen einen deutlichen Gewinn an Produktivität und Qualität durch den Einsatz der JooMDD-Infrastruktur. Weiterhin werden Design und Beobachtungen eines semi-kontrollierten Tutorials mit Entwicklern aus der Joomla-Community präsentiert. Unter Verwendung der JooMDD-Infrastruktur in mehreren Entwicklungsszenarien wurde qualitatives Feedback über die Akzeptanz, Nützlichkeit sowie offene Herausforderungen des vorgestellten MDE-Ansatzes gesammelt. Abschließend werden Lessons Learned sowie auftretende Validitätsgefährdungen der durchgeführten Studien präsentiert.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Challenges	3
1.3	Contributions	5
1.4	Methodology	6
1.5	Outline	7
1.6	Thesis-Related Publications and Presentations	8
2	Web Content Management Systems	11
2.1	WCMS Features	12
2.1.1	Sections and Roles	12
2.1.2	Core Features	13
2.1.3	Extensibility and Programmability	15
2.2	Commonalities and Differences in WCMS Extension Development	17
2.2.1	WCMS Market	17
2.2.2	Extension Development Comparison	20
2.3	Common Extension Development Scenarios	32
2.3.1	Scenario 1: Development of Independent Extensions	32
2.3.2	Scenario 2: Development of Dependent Extensions	33
2.3.3	Scenario 3: Migration of a Legacy Extension to a new Platform (Version)	34
2.3.4	Further Scenarios	36
3	Model-Driven Engineering	39
3.1	Terminology	40
3.2	MDE in Software Development	42
3.3	MDE Infrastructure Development	42
3.3.1	Design of Domain-Specific Languages	43
3.3.2	Providing Transformations	44
3.3.3	Support for MDE Tool Development	47
3.3.4	IDE Integration	48
3.3.5	Custom Code Integration	49
3.3.6	Development Process	49
4	MDE of WCMS Extensions - General Solution Concept and Requirements	53
4.1	Interviews with Extension Developers	53
4.1.1	Set-up	53
4.1.2	Results	55
4.1.3	Interpretation	62
4.2	MDE Concept for WCMS Extensions	63
4.2.1	Model-Driven Engineering of Independent Extensions	65
4.2.2	Model-Driven Engineering of Dependent Extensions	66
4.2.3	Model-Driven Migration of Legacy Extensions	68
4.2.4	Additional Scenarios	70

4.3	MDE Infrastructure Requirements Elicitation	72
4.3.1	Domain-Specific Language	73
4.3.2	Model Editors	76
4.3.3	Code Generator	78
4.3.4	Reverse Engineering Facilities	81
4.3.5	General Infrastructure Requirements	82
4.3.6	Summary	83
4.4	Discussion	83
4.4.1	Relevance of Scenarios 1-5 (RQ1.1)	84
4.4.2	MDE Concept for Scenarios 1-5 (RQ1.2)	85
4.4.3	MDE Infrastructure Requirements for Scenarios 1-5 (RQ1.3)	85
5	Domain-Specific Language for WCMS Extensions	87
5.1	State of the Art	87
5.2	Language Design	88
5.2.1	Data Modelling	92
5.2.2	Interaction Modelling	94
5.2.3	Extension Modelling	97
5.2.4	Core Support	101
5.3	Well-formedness Rules	103
5.3.1	Data Modelling	104
5.3.2	Interaction Modelling	106
5.3.3	Extension Modelling	107
5.4	Model Editors	108
5.5	Showcase Models	109
5.6	Evaluation	112
5.6.1	Requirement Verification and Validation	113
5.6.2	Adequacy of the DSL	114
5.6.3	Threats to Validity	115
6	Transformation Tools	117
6.1	State of the Art	117
6.1.1	Translation to Extension Code	118
6.1.2	Handling of Legacy Extensions	119
6.2	Code Generation of WCMS Extensions	120
6.2.1	Concept	121
6.2.2	Joomla-specific Extension Generator	124
6.3	Extraction of Deployed WCMS Extensions	141
6.3.1	Concept	143
6.3.2	ExtPorter: A Joomla-Specific Component Extractor	143
6.4	Model Extraction of Legacy Extensions	145
6.4.1	Concept	145
6.4.2	JExt2eJSL: Model Extraction of Joomla 3 Components	146
6.5	Evaluation	149
6.5.1	Requirement Verification and Validation	149
6.5.2	Threats to Validity	158
7	MDE of WCMS Extensions - Quantitative and Qualitative Analysis	159
7.1	State of the Art	159

7.2	Quantitative Analysis - Conducting a Controlled Experiment	161
7.2.1	Method	161
7.2.2	Results	165
7.2.3	Discussion	172
7.3	Qualitative Analysis - MDE Workshop with Industrial Practitioners	173
7.3.1	Method	173
7.3.2	Observations	176
7.3.3	Discussion	177
7.4	Lessons Learned	178
7.5	Threats to Validity	179
7.5.1	Construct Validity	179
7.5.2	Internal Validity	180
7.5.3	Conclusion Validity	180
7.5.4	External Validity	180
8	Summary and Outlook	183
8.1	Summary	183
8.2	Outlook	185
A	Semi-Structured Expert Interview	189
B	Meta-Model of the cJSL DSL	209
C	Well-Formedness Rules for eJSL	213
D	Generator Scalability Tests	219
E	Controlled Experiment: Documents	225
F	Controlled Experiment: Results	251
G	Hands-on Tutorial	261
	Bibliography	269

List of Figures

1.1	WCMS usage throughout the last years (based on [251])	1
1.2	Web Shop Extension within a WordPress-based Website [261]	2
1.3	Different Extension Types on a Joomla-based Website [133]	2
1.4	Gartner Hype Cycle [76]	5
1.5	Outline of this Thesis	7
2.1	Composition of a WCMS-based Web Application	12
2.2	Aggregation of Articles as Index Page in Joomla 4 Administration	14
2.3	Edit Interface for Blog Posts within WordPress 5 Administration Section	14
2.4	Concept of Installable WCMS Extensions	16
2.5	WCMS market share (based on [252], stand October 2019)	18
2.6	Module representation within a running Drupal instance (administrator view)	28
2.7	Development of Independent Extensions	33
2.8	Development of Dependent Extensions	33
2.9	Common Dependencies Between Joomla Extensions (cf. [186])	34
2.10	Migration of an Extension to a new Platform (Version)	34
2.11	Differing file structure of a Joomla 3 (left) and a Joomla 4 (right) component (administrator folder)	35
2.12	Partial Augmentation of a Legacy Extensions	36
2.13	Modernization of a Legacy Extensions	37
3.1	Relationship between MDA, MDD, MDE, and MBE (cf. [28])	40
3.2	Abstraction Level Specification of MDA	40
3.3	Template-based Code Generation (cf. [221, p. 189])	45
3.4	MDE Tools based on EMF [38]	47
3.5	Domain Engineering and Application Engineering (adapted from [84])	50
3.6	Domain Analysis Concept (adapted from [221])	51
3.7	Agile MDE Infrastructure Development Process in Action (adapted from [244])	52
4.1	Standard View with CRUD Functionality within a WCMS Extensions	54
4.2	MDE Infrastructure Concept for WCMS Extension Development (cf. [28])	64
4.3	Model-Driven Development of Independent Extensions	65
4.4	Model-Driven Development of Dependent Extensions	67
4.5	Forward, Reverse and Reengineering [54, p. 9]	69
4.6	Model-Driven Migration of Legacy Extensions	70
4.7	Model-Driven Augmentation of Legacy Extensions	71
4.8	Model-Driven Modernization of Legacy Extensions	72
4.9	Interaction between different representations of various data	74
5.1	Overview of the eJSL Meta-Model	89
5.2	Parameters of a Joomla 4 component (Users Component)	90
5.3	List and Details View of an installed Conference Component	91
5.4	List View of an installed Conference Component (Frontend)	91
5.5	Data Modelling Part of eJSL	92
5.6	Representation of an Entity Reference to another Entity Attribute	94
5.7	Interaction Modelling Part of eJSL	95
5.8	Filters for Existing Participants (J4 Backend)	96
5.9	Extension Modelling Part of eJSL	97

5.10	Concept of eJSL and cJSL Integration (cf. [183])	101
5.11	cJSL Meta-Model for Concrete WCMS Instances (Excerpt)	102
5.12	Core Support based on eJSL Models	102
5.13	Generalization Cycle between Entities	104
5.14	Reference Cycle between Entity Attributes	106
5.15	Link Attribute Must Exist in Referenced Entity	106
5.16	eJSL Model Editor within Eclipse IDE	108
5.17	eJSL Model Editor within PhpStorm IDE	109
5.18	Project Wizard within PhpStorm IDE	109
5.19	Web IDE Comprising the eJSL Model Editor	110
5.20	Showcase Model - Conference	110
5.21	Showcase Model - Shop	111
6.1	Generator Front-End including the Decorator API and Resource Transformer . .	122
6.2	Generator Back-End including the WCMS-specific Code Templates	123
6.3	Preprocessing of many-to-many Relationships between two Entities	126
6.4	Generated Language Files and File Contents	129
6.5	Architecture of Joomla Components	130
6.6	Automatically Created Component Link as Menu Item in the Administration Sec- tion of a Joomla Instance (J3)	130
6.7	Generated File Structure of a Joomla 3 (left) and 4 (right) Component	131
6.8	Frontend Details View of a generated Component	132
6.9	Form Field for Referenced Attributes	133
6.10	Additional Tab in Edit View with Form Field for Multiple References	134
6.11	Representation of Multiple References in a List View	134
6.12	Architecture of Joomla Modules	136
6.13	Generated File Structure of a Joomla 3 (left) and 4 (right) Module	137
6.14	Representation of a Generated Module illustrating the Data of an installed Com- ponent in the Frontend of a Joomla Instance)	138
6.15	Configuration of a Generated Module in the Administration Section	138
6.16	Architecture of Joomla Plugins	139
6.17	Generated File Structure of a Joomla 3 and 4 Plugin	139
6.18	Architecture of Joomla Templates and Libraries	140
6.19	Generated File Structure of a Joomla Template and Library	140
6.20	Separation of a Deployed Joomla Component	142
6.21	Extension Extraction Concept	143
6.22	Details View: Create a new Extracted Component	144
6.23	Details View: Installable Component Package as Zip File	144
6.24	List View of the ExtPorter Component	145
6.25	Model Extraction Concept	146
6.26	Overview of the JExt2eJSL Architecture	146
6.27	Extension Parsing Process	147
6.28	List View for Pre-Course Management	150
6.29	Edit/Details View for Pre-Course Management	150
6.30	Generated File Structure of Partial Update Code (left) and Installable Component Code (right)	152
6.31	Scalability Test 1 (Components)	153
6.32	Scalability Test 2 (Pages)	154
6.33	Scalability Test 3 (Entities with References)	154
6.34	Scalability Test 4 (References in one Entity)	155
6.35	Scalability Results of all Tests	155
6.36	Gradle Tasks for Build Automation of JooMDD Plugins	156

7.1	Entity Model of Possible Solution for CRM Requirement	163
7.2	Procedure Overview	165
7.3	Passed Test Case Ratio (Overview)	166
7.4	Q-Q Plot for Productivity Result Sets (Traditional/MDE)	168
7.5	Code Style Violations / LoC Ratio (Overview)	170
7.6	Q-Q Plot for Quality Result Sets (Traditional/MDE)	172
7.7	Tutorial Procedure Overview	176
B.1	cJSL Application with all cJSL Parts	209
B.2	cJSL Configuration Part	209
B.3	cJSL User Part	210
B.4	cJSL Menu Part	210
B.5	cJSL Content Part	211
B.6	cJSL Page Part	212
D.1	Measurement Setting (Test 1)	219
D.2	Measurement Setting (Test 2)	220
D.3	Measurement Setting (Test 3)	221
D.4	Measurement Setting (Test 4)	222
D.5	Measurement Setting (Test 5)	223
E.1	Consent Form	225
E.2	Experiment Presentation (1)	226
E.3	Experiment Presentation (2)	227
E.4	Experiment Presentation (3)	228
E.5	Experiment Presentation (4)	229
E.6	Experiment Presentation (5)	230
E.7	Experiment Presentation (6)	231
E.8	Experiment Presentation (7)	232
E.9	Experiment Presentation (8)	233
E.10	Experiment Presentation (9)	234
E.11	Demographic Questionnaire	235
E.12	Self-Assessment (Self-Assessment: General Software Development and Joomla)	236
E.13	External Assessment: Joomla Knowledge)	237
E.14	Self-Assessment (Self-Assessment: MDE)	238
E.15	Self-Assessment (External Assessment: MDE)	239
E.16	Questionnaire after Session 1	240
E.17	Questionnaire after Session 2	240
E.18	Feedback Questionnaire	241
E.19	Requirement A: University Management	242
E.20	Requirement A: Test Cases (1)	243
E.21	Requirement A: Test Cases (2)	244
E.22	Requirement A: Test Cases (3)	245
E.23	Requirement B: Customer-Relationship Management	246
E.24	Requirement B: Test Cases (1)	247
E.25	Requirement B: Test Cases (2)	248
E.26	Requirement B: Test Cases (3)	249
G.1	Hands-on Tutorial Presentation (1)	261
G.2	Hands-on Tutorial Presentation (2)	262
G.3	Hands-on Tutorial Presentation (3)	263
G.4	Hands-on Tutorial Presentation (4)	264
G.5	Hands-on Tutorial Presentation (5)	265

G.6	Hands-on Tutorial Presentation (6)	266
G.7	Hands-on Tutorial Presentation (7)	267
G.8	Hands-on Tutorial Presentation (8)	268

List of Tables

1.1	Number of officially listed extensions for current WCMS versions (Sept. 2020)	3
2.1	Developer Support	29
2.2	Extension Features (1)	29
2.3	Extension Features (2)	30
2.4	API Support	30
2.5	API Support (2)	31
2.6	Scoring of WCMS Extensibility	31
4.1	Experience with Joomla (C1)	55
4.2	Development of Joomla/WCMS Extensions (C2)	56
4.3	Migration of Joomla/WCMS Extensions (C3)	57
4.4	Extension Characteristics (C4)	58
4.5	MDE Approach during Extension Development/Migration (C5.1)	59
4.6	MDE Approach during Extension Development/Migration (C5.2)	61
4.7	MDE Infrastructure for WCMS Extensions - Requirements	84
6.1	Tool support for WCMS Extension Development Scenarios (for Joomla 3)	118
6.2	Type Mappings for eJSL Standard Types and HTML Types	126
6.3	Type Mappings for eJSL Standard Types and SQL Types	134
6.4	Type Mappings for eJSL HTML Types	135
7.1	Study Design	163
7.2	Productivity Results: Overview (Amount of passed Test Cases)	166
7.3	Productivity Results: Detailed Insights (Component Structure)	167
7.4	Productivity Results: Detailed Insights (Component Views)	167
7.5	Productivity Results: Detailed Insights (Component CRUD)	168
7.6	Productivity Results: Detailed Insights (Module)	168
7.7	Quality Results: Overview (Violations/LoC)	169
7.8	Quality Results: Detailed Insights (Component Views: List)	170
7.9	Quality Results: Detailed Insights (Component Views: Edit)	171
7.10	Quality Results: Detailed Insights (Module)	171
D.1	Measurement Setting (Test 1)	219
D.2	Measurement Result (Test 1)	220
D.3	Measurement Setting (Test 2)	220
D.4	Measurement Result (Test 2)	221
D.5	Measurement Setting (Test 3)	221
D.6	Measurement Result (Test 3)	222
D.7	Measurement Setting	222
D.8	Measurement Result (Test 4)	223
D.9	Measurement Setting (Test 5)	223
D.10	Measurement Result (Test 5)	224
F.1	Developer Experience (1)	251
F.2	Developer Experience (2)	252
F.3	Experience with a WCMS and Joomla	252
F.4	Experience with MDE	253

F.5	Open-Mindedness towards MDE	253
F.6	Session 1: Productivity Results (Row Data)	254
F.7	Session 1: Quality Results (Row Data)	254
F.8	Session 1: Session Feedback (Row Data)	255
F.9	Session 1: Session Feedback - Development Approach (Row Data)	255
F.10	Session 2: Productivity Results (Row Data)	256
F.11	Session 2: Quality Results (Row Data)	256
F.12	Session 2: Quality Results - Modules (Row Data)	257
F.13	Session 2: Session Feedback (Row Data)	257
F.14	Feedback Results - Experiment (Row Data)	258
F.15	Feedback Results - MDE (Row Data)	259

Listings

2.1	Minimum specification within the main PHP file of a WordPress plugin	22
2.2	WordPress action hook	22
2.3	Joomla plugin structure (Joomla's user plugin)	24
2.4	Joomla manifest structure (Joomla's user component)	25
2.5	Drupal configuration file (helloWorld example)	27
5.1	Data Model for Conference Extension (Inheritance and Attributes)	93
5.2	Data Model for Conference Extension (References)	93
5.3	Page Model for Conference Extension (Static Page)	96
5.4	Page Model for Conference Extension (Custom Page)	96
5.5	Page Model for Conference Extension (Index Page)	96
5.6	Page Model for Conference Extension (Details Page)	97
5.7	Extension Model for Conference Extension (Manifest)	98
5.8	Extension Model for Conference Extension (Languages)	98
5.9	Extension Model for Conference Component	99
5.10	Extension Model for Conference Module	99
5.11	Extension Model for Conference Search Plugin	100
5.12	Extension Model for Conference Library	100
5.13	Extension Model for an Example Template	101
5.14	Core Model for WCMS User Management	103
5.15	Extension Model with Reference to the WCMS Core	103
5.16	Constraint for Unique Attribute Identifiers (within one Entity)	104
5.17	Constraint for consistent Entity Inheritance	104
5.18	Constraint for Transitive Closure of Entity Generalization (avoid Generalization Cycle)	105
5.19	Constraint for Auto Increment Property	105
5.20	Constraint for Multiplicity Values	105
5.21	Constraint for Valid Multiplicity Relations (Between min and max)	105
5.22	Constraint to avoid Entity Reference Cycles	105
5.23	Constraint to Check if Datepicker (HTML Type) is Mapped to Time, Date and Datetime (Entity Type)	106
5.24	Constraint to Check if Linked Attribute in IndexPage is Consistent to Referenced Entity Attribute	107
5.25	Constraint to Check if Table Columns are Consistent to Referenced Entity in IndexPage	107
5.26	Constraint to Check the Consistency of Multiple Referenced Entities	107
5.27	Constraint to Check Unique Extension Names (Same Extension Kind)	108
5.28	Generic Showcase Model with Placeholders (Excerpt)	112
6.1	Example of Page Definition before Preprocessing Step	127
6.2	Example of Page Definition after Preprocessing Step	127
6.3	Language Specification for an Extension in an eJSL Instance Model	129
6.4	Entity Definition with unidirectional Reference	132
6.5	Entity Definitions with bidirectional Reference (Participant \longleftrightarrow Talk)	133
6.6	Field Definition in A Form Specification File	136
6.7	Module Dependency to a Model of a Component	137
6.8	Generated Search Plugin Class (Excerpt)	139
6.9	Module Dependency to a Page of a Component	141
6.10	Component and Dependent Module as Part of an Extension Package	141
6.11	Placeholder within an Extracted Extension Model	148

C.1	Well-Formedness-Rules: Context Entities	213
C.2	Well-Formedness-Rules: Context StandardTypes	213
C.3	Well-Formedness-Rules: Context Reference	214
C.4	Well-Formedness-Rules: Context Feature	214
C.5	Well-Formedness-Rules: Context DetailsPage (1)	214
C.6	Well-Formedness-Rules: Context DetailsPage (2)	215
C.7	Well-Formedness-Rules: Context IndexPage	216
C.8	Well-Formedness-Rules: Context Library	216
C.9	Well-Formedness-Rules: Context Class	216
C.10	Well-Formedness-Rules: Context CMSExtension	217

*Research is to see what everybody else has seen,
and to think what nobody else has thought.*

– Albert Szent-Gyorgyi

Development in the web domain has changed tremendously throughout the last years. Web applications are no longer static HTML documents with the intention of just providing information. Nowadays, web applications are functional rich applications, which are no longer distinguishable from native applications. They are used within various domains for almost every purpose. To this end, the web domain consists of the most fluctuating and growing technologies nowadays. Besides conventional client-server development (by using HTML, CSS and JavaScript), a massive set of tools and frameworks exist to support web developers during the creation of functional rich web applications. Especially the rise of server-side JavaScript for backend development using the Node.js framework [68] as well as JavaScript-based frameworks for frontend development, such as Angular [79] and React [63], became very popular throughout the last years.

Among these frameworks, web content management systems (WCMSs) established as the most popular choice for creating a web application. During the last decade, WordPress [266], Joomla [171], and Drupal [34] stand out as the most popular choice for creating a WCMS-based web application. The purpose of a WCMS platform is to provide certain core functionalities such as management of users, content, sites, media, templates, and languages. Using a WCMS typically simplifies the creation of a web application, since it usually can be configured within a simple installation dialogue. This even allows non-developers to create a web application.

In 2011 only 25% of all websites were based on a WCMSs (top 5: 19%), whereas in 2020 the majority of all websites are based on a WCMS (more than 57%, top 5: 43% [252]).

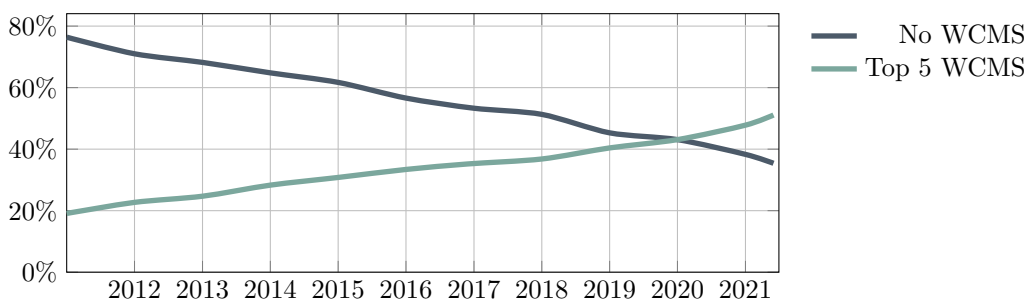


Figure 1.1: WCMS usage throughout the last years (based on [251])

1.1 Motivation

Most of the common WCMSs provide the possibility of functional augmentation by installable software extensions. These software extensions can vary from simple widgets for data representation at any website position up to functional rich stand-alone extensions, which are used for

the management of whole domain objects. This allows, e.g., the integration of a media manager or a web shop into a WCMS instance without changing the platform itself. So, some valuable benefits can be ensured. E.g., leaving the platform unchanged guarantees consistency of the system, even after version updates. Additionally, the encapsulation of function implementations as extensions supports the reuse and free distribution of functionality.

The following figures present the typical appearance of WCMS extensions within WCMS-based websites. In Chapter 2 we will take a closer look at the differences between the mentioned WCMSs and their functional extensibility.

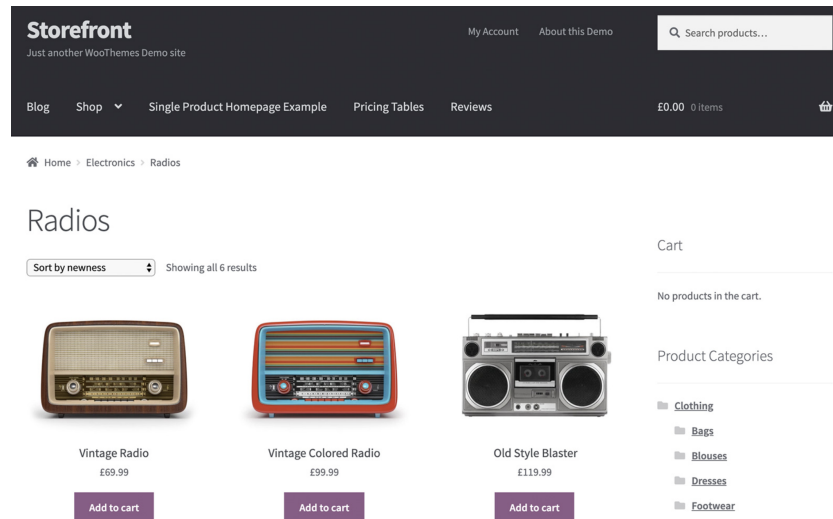


Figure 1.2: Web Shop Extension within a WordPress-based Website [261]

Figure 1.2 illustrates the use of a web shop plugin (WooCommerce [261]) within a WordPress-based website. With the use of this extension, the administrator of the site is able to provide a web shop, even though the basic core of WordPress is intended to manage blog posts.

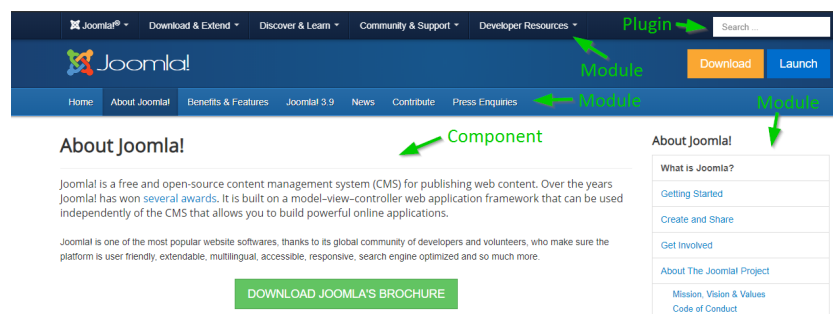


Figure 1.3: Different Extension Types on a Joomla-based Website [133]

Figure 1.3 illustrates how the more sophisticated extension mechanism of Joomla is used within a Joomla-based website. To ensure homogeneous integration of extensions into the host system, Joomla provides various extension types which can be used within a Joomla instance. The two most frequently used extension types are components and modules. Components can extend the functionality of a Joomla-based site tremendously, since they typically consist of own database tables and allow the management of custom data entities. This usually exceeds the typical functional scope of extensions which are used within other WCMSs. Modules typically represent data which is managed by one or more components within module positions on a Joomla instance.

Such interdependencies are also common between extensions of other WCMSs like WordPress and Drupal, allowing extension developers to *augment already existing extensions by custom features* and is therefore a popular procedure.

A look at the current extension statistics of the most popular WCMSs (Table 1.1) indicates their relevance for WCMS-based websites. The statistic contains only the number of extensions which are listed in the official extension directories of the respective WCMS. Without the use of extensions, WCMSs would have hardly been so successful over the last years.

Table 1.1: Number of officially listed extensions for current WCMS versions (Sept. 2020)

WCMS	Versions	# of Existing Extensions
WordPress [266]	all versions	57.487
Joomla [170]	3.x, 4.0 beta	7.449
Drupal [34]	9.x	3.906

To support extension developers, the most common WCMSs provide an API for using the WCMS as framework. This decreases the amount of extension code and allows a simplified deployment. Usually, extensions can be deployed to a running WCMS instance by using an installation dialogue within the WCMS instance. However, this requires the implementation of prescribed API functions. Though, *developing against the API of a WCMS requires technical knowledge and is only helpful for advanced extension developers*.

A closer look to the required code structure points out the typical side effect of standardized API code - a *tremendous amount of boilerplate code*. Due to e.g. Joomla's architecture, code of the same structure has to be repetitively developed in accordance with the required development standards. These in turn can require strange rules such as same identifiers for class and file names to ensure correct interaction between the core platform and the installed extension. Novices in the domain are often overstrained leading them to develop extensions without following the API guidelines or to apply the clone-and-own¹ approach - developers copy and adapt existing code to new requirements without an actual understanding of the code. This error-prone procedure often leads to *low quality extensions which are hard to maintain*.

1.2 Challenges

WCMS extension developers face several challenges during common development scenarios. These scenarios include the *initial and further development of independent and dependent extensions* as well as the *migration of existing extension code to new platforms* including new major versions of the same platform. Especially *developers with less experience must invest immense effort to implement or maintain standard extensions of high quality*.

Usually, a tremendous amount of generic code fragments have to be implemented for every popular WCMS in order to adhere to their coding standards and using their API properly. A common procedure for extension developers is to make a copy of existing extension code which is then adapted to the new requirements (clone-and-own practice). However, this is an error-prone practice, e.g. if developers have lacking knowledge of the existing code which is used as baseline code [125, 124, 268]. In our context, this often leads to oversized extensions which contain useless code or features which are not required. This effect is intensified, if code from different extensions is merged into a new extension. To ensure high extension quality, requires refactorings of the new extension, provoked by the development practice.

¹The clone-and-own approach is a common practice in software product line (SPL) development where software variants are part of software families [48]. This ensures high reusability of software features during the derivation of new product variants.

⇒ *Problem Statement 1:* Ensuring high quality in WCMS extensions requires tremendous effort due to required coding guidelines and APIs which have to be implemented.

Typically, WCMSs go through several version changes within their life cycle (see Section 2.2.1). According to the last major version releases of the most popular WCMSs, these changes are usually accompanied by architectural modifications in the code base of the WCMS platform. *Extension developers are forced to migrate their extensions to the new versions to ensure their operability within updated WCMS instances, which is a time-consuming process.* A high number of extensions to migrate typically leads to escalating maintenance effort. As experience has shown, *missing documentation and required effort often lead to dying extensions* since developers were not able to migrate their software in a proper way. If this happens, administrators have to replace the extension which in turn is associated with additional effort for them. Since WCMS extensions may include dependencies to other extensions, a missing migration typically affects depending extensions as well.

⇒ *Problem Statement 2:* The code migration of existing extensions to new platform (versions) requires tedious effort, especially if the number of extensions to migrate rises.

Another challenge faced by extension developers occurs, if WCMS administrators make use of third-party or legacy extensions in a WCMS instance which have to be functionally augmented. Maintainers of third-party extensions are not always willing to augment their extensions. So, legacy extensions are typically not further maintained by their initial developers but may comprise valuable features for WCMS administrators. Therefore, it is common practice to augment (legacy) extensions by new features or re-implement the extension, e.g. if it must be migrated to a new major platform version. Both requires a reverse engineering process which must be tediously performed by extension developers.

⇒ *Problem Statement 3:* The augmentation and re-engineering of (legacy) extensions requires a time-consuming reverse engineering process.

Managing dependencies between extensions is a challenging task, since every evolution of the dependency could break the dependent extension. Due to the missing dependency management in popular WCMS frameworks, administrators have to ensure proper interplay between depending extensions in a WCMS instance. Extension developers must react to each dependency update and migrate dependent extensions to new versions of the dependency - a tedious task, if the dependency is maintained by different developers.

⇒ *Problem Statement 4:* The maintenance of dependent extensions is tedious due to the missing dependency management between extensions.

To address the high amount of repetitive code, developers of WCMS extensions often publish empty boilerplate code or boilerplate generators to create an initial extension. However, they can only be used for initial development and do not support developers during further development of the extensions. Usually the boilerplate code contains only exemplary excerpts of the required artefacts such as model, view or controller files. If a developer requires more than one of these artefacts he must develop them by hand (usually by copy&paste). So, *boilerplate solutions are helpful for the initial creation of an extension but are not very helpful during the further development of an extension.* Especially, *if a new extension has to be developed which augments an existing legacy extension, developers have to understand and manually incorporate dependencies tediously.* The same applies to extensions which have to be migrated to a new major platform version or a different WCMS platform. This scenario is not supported by current tool support.

⇒ *Problem Statement 5:* Existing tool support does not support iterative extension development, augmentation of existing legacy extensions, or extension migration to new platforms.

1.3 Contributions

In order to support WCMS extension developers during the development and maintenance of WCMS extensions, developers should make use of a sophisticated development practice. The current practices in the domain, clone-and-own and using boilerplate code, are not sophisticated and provoke additional maintenance effort for developers. In accordance to [160], there is “little recent research literature reporting on modern web development practices, especially concerning the use of platforms such as WordPress”.

A promising development practice is represented by model-driven engineering (MDE). By using models as reusable artefacts developers can be protected from the technical knowledge. This in turn is placed within code generators which use these models as input and generate the most parts of the application. Due to the ability of using models in different (versions of) code generators, code can be simultaneously generated for different platforms or versions of the same platform. Therefore, MDE should be researched as an alternative development approach, to investigate its usefulness during WCMS extension development.

In 2012 Gartner researched the hype [76] of model-driven architecture (MDA²) among other emerging technologies. Gartner came to the result, that MDA is on its way to the point of disillusionment [13]. However, in accordance to [28], it passed the point of disillusion and is currently passing the "Slope of Enlightenment" on its way to the "Plateau of Productivity". To reach this plateau requires evidence for its productivity within different domains. We assume that the WCMS domain is one of the domains where an MDE adoption can increase productivity.

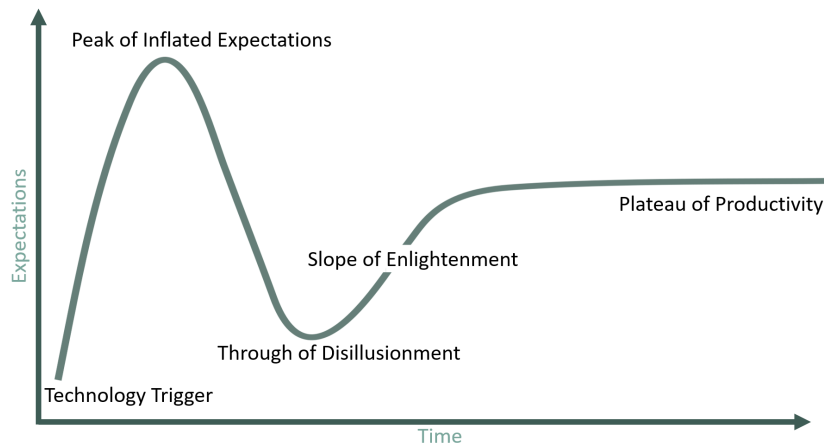


Figure 1.4: Gartner Hype Cycle [76]

General MDE approaches as the ones described in [27], [137], [42], and [225] can be used to create complete websites in a model-driven manner. However, the proposed approaches and tools are not suitable for our research, since they do not address WCMSs and their extensions. The reason for this is very straightforward: The use of WCMSs as dynamic web applications is a relatively new procedure which has been established throughout the last decade.

To address the previously defined problem statements during common development scenarios, we propose an MDE approach to WCMS extension engineering instantiated at the WCMS Joomla. By applying this approach to WCMS extension development, we study its profitability to find evidence for the following theory: **A model-driven engineering approach is profitable in the WCMS domain.**

²MDA is a formal specification of MDE and was typically used as the main term for any MDE adoption in industry.

So, the fundamental goal of this work can be achieved by showing that a model-driven approach is supportive in the domain of WCMSs (better quality, less complex development) and is a profitable (faster) alternative to common development practices in the WCMS domain. In this work, the following contributions are provided:

- *Overall MDE concept for common WCMS extension development scenarios.* This includes the comparison of extension development for the most popular WCMSs, the elicitation of common extension development scenarios, and the confirmation of these scenarios by industrial practitioners from the WCMS domain. The stressed scenarios include initial and continuous engineering scenarios also for existing legacy extensions in running systems.
- *Development of an MDE infrastructure for WCMS extensions based on a conducted requirements elicitation.* This includes a platform-independent domain-specific modelling language for the abstract description of WCMS extensions, model editors, and showcase models which can be used e.g. for the creation of prototype extensions. In addition, the infrastructure consists of a platform-specific code generator supporting extension development of two major versions of the Joomla WCMS. Moreover, platform-specific reverse engineering facilities for the same WCMS are included.
- *Evaluation of the profitability of MDE for the development of WCMS extensions.* This evaluation is based on two empirical user studies with novice and expert extension developers of the WCMS community.

In this work, we concentrate completely on the development of software extensions for WCMSs by applying a model-driven approach. This leads to some limitations. Since there is little related research in the problem domain we have to concentrate on the main challenges to obtain first results about the applicability and profitability of the approach. We mainly focus on the WCMS Joomla as first reference system expecting similar research results for other WCMSs. Even though we investigate the suitability of the approach in other WCMSs such as WordPress and Drupal, we only scratch the surface of the generalization of the approach. Further research should focus on generalizability of our findings.

Another limitation is given by the used framework for the infrastructure development. Currently, the Eclipse Modeling Framework (EMF), combined with Xtext and Xtend, is the state of the art in the development of MDE infrastructures. So, we do not investigate infrastructures developed with alternative frameworks. However, we will introduce alternatives with related references in Section 3.3.

Moreover, we consider the common development scenarios regarding the development life cycle of WCMS extensions. Though, we do not investigate new methods or tools for the life cycle of the MDE infrastructure itself. This has already been done in early stages of MDE throughout the last decade (cf. [72]). Another limitation of this work is the management of individual code within model-driven development scenarios. Nevertheless, we will discuss possible integrations in Chapter 3.

1.4 Methodology

In this work we follow a straightforward approach to research the applicability and profitability of an MDE approach during common extension development scenarios. In this context, we address the following general research questions:

RQ1: How can MDE support common WCMS extension development scenarios?

RQ2: To which extend can MDE support WCMS extension developers during development and maintenance of WCMS extensions?

To address RQ1, we *collect the common features of WCMSs and compare the most popular systems to identify the equalities and differences between the systems and ensure an appropriate choice of a reference system*. We use one specific WCMS as reference in a first phase in order to generalize the findings to other WCMSs to ensure the external validity of the approach. Based on this, we begin with the *identification of relevant development scenarios* in the domain. For this purpose, we specify common scenarios and confirm their relevance as well as the typical development challenges by conducting a set of interviews with domain experts, in our case practitioners from the WCMS community. For these scenarios, we *define a general solution concept for the integration of MDE infrastructure components within model-driven process recommendations*. Based on these concepts and the results of the conducted expert interviews, we *collect MDE infrastructure requirements* based on a research of common WCMSs such as WordPress, Joomla and Drupal.

In order to address RQ2, an MDE infrastructure is developed as basis for a comparison of MDE with conventional extension development. Based on the previously elicited requirements, we discuss existing DSLs and their suitability for modelling WCMS extensions. Building on this, *a platform-independent DSL is developed* in addition to well-formedness rules, model editors, and showcase models. To gain insights about the profitability of MDE during WCMS extension development we *define concepts for MDE transformation tools such as code generators and model extractors and implement platform-specific prototypes* based on appropriate reference extensions for a reference WCMS - in our case the Joomla WCMS. To ensure high quality of these tools, we follow an iterative development method and adhere to reference extensions which follow the Joomla guidelines.

By applying our MDE infrastructure during the confirmed development scenarios in the next phase, we *compare conventional WCMS extension development with model-driven engineering*. To this end we conduct a controlled experiment with the focus on the development of independent and dependent extensions as first iteration to allow conclusions on the general suitability of a model-driven approach in the WCMS domain. We quantitatively research the impact of MDE in terms of the development speed during WCMS extensions development and the quality of WCMS extensions. In addition, we investigate, if it is possible for inexperienced developers to develop a WCMS extension by using an MDE infrastructure.

Moreover, we *apply the MDE infrastructure during common extension development scenarios* in a semi-controlled tutorial with industrial practitioners from the Joomla community to gain insights of its appropriateness during these scenarios in a qualitative manner. By applying the approach within real-world development projects throughout the chosen development use cases, we contrast to typical academic research in the same domain.

1.5 Outline

As Figure 1.5 illustrates, this work is structured as follows.

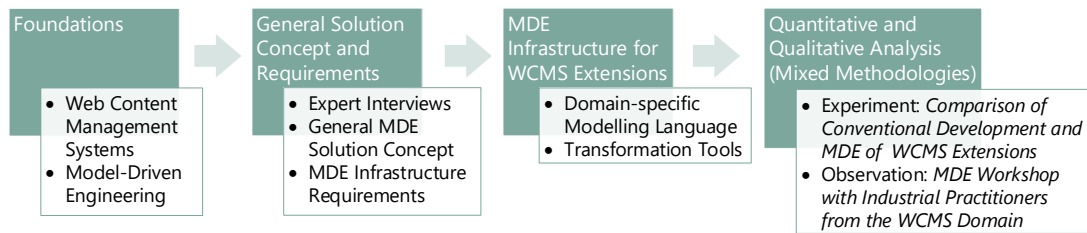


Figure 1.5: Outline of this Thesis

In Chapter 2, we outline the fundamentals of web content management systems and their extensibility through installable software extensions. This includes a specialized focus on their development challenges and a comparison of popular WCMSs in order to identify a suitable reference system for the initiation of our research. Additionally, the elicitation of common extension development scenarios is placed in this chapter.

Subsequently, we give an overview of the common terminology of model-driven engineering and present common MDE infrastructure artefacts in Chapter 3. This also includes the presentation of popular frameworks and development methods for MDE infrastructure development.

Chapter 4 presents the general MDE concept and requirements for WCMS extension development scenarios. This also includes the presentation of the results of our conducted expert interviews with industrial practitioners. The presented concepts are a proposed solution for the previously described problem statements 1-4.

Based on the collected MDE infrastructure requirements, we present a domain-specific modelling language for WCMS extensions in Chapter 5 and transformation tools for WCMS extensions in Chapter 6. Both chapters comprise a discussion of the respective current state of the art, general concepts, examples, and a conclusive evaluation. With the presented infrastructure, the concepts as defined in Chapter 4 can be realised in order to address problem statements 1-4. Furthermore, the presented tools represent a necessary extension of the existing tool landscape for the iterative development of WCMS extensions, with which we directly address problem statement 5.

The controlled experiment which compares conventional with model-driven WCMS extension engineering is presented in Chapter 7. This includes a state of the art discussion of empirical studies on MDE in practice, the introduction of our study designs, results and interpretation, as well as an evaluation considering the validity threats of our studies.

With Chapter 8, we summarize and evaluate our work followed by an outlook on further research.

1.6 Thesis-Related Publications and Presentations

The following list of papers are a collection of related contributions addressing this thesis (in chronological order):

- [183] D. Priefer. Model-driven development of content management systems based on Joomla. In I. Crnkovic, M. Chechik, and P. Grünbacher, editors, *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 911–914, New York, 2014. ACM.
- [188] D. Priefer, P. Kneisel, and G. Taentzer. JooMDD: A Model-Driven Development Environment for Web Content Management System Extensions. In *ICSE Companion '16: Companion Proceedings of the 38th International Conference on Software Engineering*, pages 633–636, New York, NY, USA, 2016. ACM.
- [187] D. Priefer, P. Kneisel, and G. Taentzer. A Model-Driven Process to Migrate Web Content Management System Extensions. In A. Bozzon, editor, *Web engineering*, volume 9671 of *Lecture Notes in Computer Science Information systems and applications, incl. Internet/Web, and HCI*, pages 603–606, Cham and Heidelberg, 2016. Springer.
- [186] D. Priefer, P. Kneisel, and D. Strüber. Iterative Model-Driven Development of Software Extensions for Web Content Management Systems. In A. Anjorin and H. Espinoza, editors, *Modelling Foundations and Applications: 13th European Conference, ECMFA 2017, Held as Part of STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings*, pages 142–157, Cham, 2017. Springer International Publishing.

- [185] D. Priefer, P. Kneisel, W. Rost, D. Strüber, and G. Taentzer. Applying MDD in the Content Management System Domain: Scenarios and Empirical Assessment. In *ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems*, pages 56–66. IEEE, 2019.
- [190] D. Priefer, W. Rost, D. Strüber, G. Taentzer, and P. Kneisel. Applying MDD in the content management system domain: Scenarios, tooling, and a mixed-method empirical assessment. *Software & Systems Modeling*, 2021.

All conference papers were presented by the author of this thesis.

The developed MDE infrastructure (JooMDD), which is presented in this work, was presented at the following national and international Joomla conferences and developer meetings (in chronological order):

- D. Priefer.** JooMDD - Model-driven Development Environment for Joomla Extensions. Presentation at the international Joomla! Conference *J and Beyond*, Prague, Czech Republic, 2015.
- D. Priefer.** JooMDD - Joomla Extension Creation for Everybody. Presentation at the *Joomla!Dagen Nederlande*, Zeist, Netherlands, 2016.
- D. Priefer.** Augmentation of existing Extensions using a Low-Code Platform. Presentation at the international Joomla! Conference *J and Beyond*, Kraków, Poland, 2017.
- D. Priefer.** Augmentation of existing Joomla! Extensions using a Low-Code Platform. Presentation at the international *Joomla! World Conference*, Rome, Italy, 2017.
- D.Priefer, W. Rost.** JooMDD Workshop . Joomla! Developer Meeting NL, Dongen, Netherlands, 2017
- D. Priefer, W. Rost.** Create a Joomla 4 Component with JooMDD. Presentation at the *Joomla!Dagen Nederlande*, Eindhoven, Netherlands, 2018.
- D. Priefer, W. Rost** Create a Joomla 4 Component with JooMDD Presentation at the international Joomla! Conference *J and Beyond*, Cologne, Germany, 2018.

The JooMDD project is hosted as open source project on GitHub: [184]

2

Web Content Management Systems

Compared even to the development of the phone or TV, the Web developed very quickly.

– Tim Berners-Lee

In today's web domain the creation of functionally rich applications from scratch is outdated and no longer state of the art. Instead, web developers use existing **Web Content Management Systems** (WCMSs) [150] which contain the main functionality of a typical web application like *user management with ACL¹ support, content management and relating menu management, media management, template management, and multi language support.*

Popular web development frameworks like Laravel [174], Laminas (Zend) [141], or CakePHP are still used for PHP-based web applications. However, with the introduction of the Node.js framework for server-side JavaScript applications [68] in addition to frameworks for front-end development like Angular [79], React [63], and Vue [267], JavaScript-based frameworks have emerged as a valuable alternative to PHP-based frameworks within the last years. Though, both PHP- and JavaScript-based frameworks require more technical knowledge and are mainly used for individual projects without the need of core functionality out of the box. Additionally, the development and maintenance of all software artefacts must be provided by the developer itself. By using a WCMS, this effort is outsourced to the respective community or provider of the WCMS. Therefore, more than half of the existing websites are based on a WCMS according to current WCMS usage statistics [251].

Since WCMSs are that popular, the web prefix is often skipped in common language use. So, whenever people talk about CMSs they most likely mean WCMSs. The domain of CMSs, however, encompasses different types of systems. WCMSs are typically used to provide web-based applications, whereas **Enterprise Content Management Systems** (ECMSs) are used within companies to manage content in order to support the organizational processes [2, 145]. The latter systems are often tailored directly to the need of the employees - typically as intranet-based systems. Usually, ECMSs are integrated into the daily work routine of employees so that they do not notice that the underlying system is a CMS. ECMS-based solutions are often integrated into other ERP² software such as SAP ERP or Oracle E-Business Suite. Most ECMSs are proprietary and commercial. This leads to a significant lower market share in comparison to open source and free-to-use WCMSs. However, they play a significant role in industry.

Another often mentioned CMS type is represented by so-called **Component Content Management Systems** (CCMSs). In contrast to WCMSs, these systems manage content as reusable components of a system, which has to be stored in a consistent and accurate way [5]. WCMSs like WordPress or Joomla manage their content in posts, pages, or articles what can lead to inconsistent or redundant content. The aim of CCMSs is to avoid inconsistencies and store the available content fragments (components) to allow their reusability. Even though popular WCMSs are no CCMSs, extensions for them are usually intended to manage content in form of reusable components or assets (see Section 2.2.1). So, a WCMS-based web application with such an extension installed can be seen as a hybrid between WCMS and CCMS.

¹Access Control List

²Enterprise Resource Planning

To complete the terminology of CMSs, **Digital Asset Management Systems** (DAMs) should be mentioned. The set of content types of DAMs is limited to digital assets (images, audio, and video) which can be used in other media [17]. Since DAMs are also used for editing the media assets, add-ons for media-related software, such as Adobe's Creative Cloud, are available for the popular systems.

2.1 WCMS Features

In this section, the main features of WCMSs will be collected. Beside the given *core functionality* like content, user, and menu management, WCMSs are characterized by their *extensibility features*. Both aspects will be examined on the following pages in an objective manner to ensure a deep understanding of WCMSs in general.

2.1.1 Sections and Roles

Typically, a WCMS-based website is divided into two accessible sub-sites (*sections*). Besides the so-called *frontend*, the section which website users access, the most WCMSs provide an additional administrator section which is usually called the *backend*. The backend is typically accessed by administrators to configure the WCMS instance and users with the rights to configure or publish content (content managers), whereas the frontend is accessed by website visitors. However, most of the systems allow some administrative actions in the frontend, too. To which extend depends on the given rights of the users.

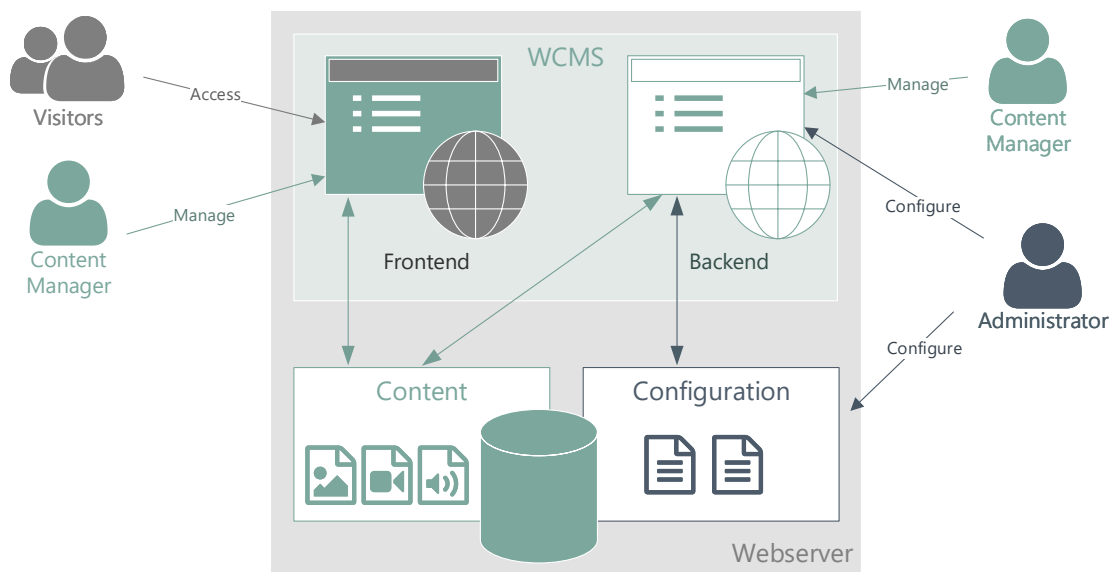


Figure 2.1: Composition of a WCMS-based Web Application

Figure 2.1 illustrates the typical composition of a WCMS-based website. Administrators can configure the WCMS via the backend. Usually, this includes the system configuration which is required for a smooth interaction between the WCMS, the web server, and a connected database. Additionally, the administrators define the menu structure for the website and create a permission concept for users and user groups. The installation of additional extensions is also typically assigned to the administrator. This also includes the installation and configuration of a basic theme/template for the frontend.

Content managers are responsible for the management of content (pages, articles, blog posts, or frontend modules). Usually, the content managers are separated into different user groups with the specific permissions to create (authors), edit (editors), or publish (publishers) content. Depending on the user group, a content manager can access the backend or has to work via the frontend of a WCMS-based website. This, however, is up to the administrators of the page.

Visitors of a WCMS-based website can access the content via the frontend of the site. Often, they can also register and login to the site to gain access to content provided for specific user groups. For instance, if a student is registered to a WCMS-based university website the student can login to get access to specific information, such as a personalized schedule or study program. Typically, visitors of a WCMS-based website do not recognize, that the site is based on WCMS, even though some indicators allow inferences to the underlying WCMS (see Section 2.2.1).

2.1.2 Core Features

In [17] the author defines the *basic and extended core features* which should be included by a WCMS. These features consist of content management features as well as features for user, multi-site, language, and media management. Within the following sub sections the relevant aspects for this work are determined more detailed based on the definitions of [17].

Basic Functionality

The basic core functionality of a WCMS obviously includes features for managing content in a comfortable way for all users. Typically, every WCMS implements a suitable *content model* which defines the structure of the managed content. The content model defines the structure of the content elements and specifies their types. Usually, the content and its representation should be separated. In this case, the content model defines the internal type definition of the content elements (e.g. the type of a text field or the required values for a content type) while they can be represented in various presentation formats. Some systems allow customization of their content model whereas others like WordPress and Joomla allow customizations through custom field definitions.

Furthermore, *embedding* and *references* between content elements can be specified in the content model³. Besides text formats, other media formats like images and videos became essential content types. Therefore, the content model must allow users to embed these formats e.g. into articles. An example for content references is the possibility of including links to articles which refer to other articles. In addition, references include content inheritance and content composition. The latter is typically implemented in form of content categories which can be a collection of sub-categories.

Another powerful basic feature of WCMSs is the aggregation of content in suitable representation formats. This includes navigation links like menu items in menus as well as result pages of a content search. Additionally, content aggregation in form of index pages which list interrelated content elements. If the content elements are ordered, such pages typically allow to reorder the elements. Another basic feature for aggregated content are filters which can be used to reduce illustrated content based on filter rules (e.g. specific content types, author, or matching search expression). The following example (Figure 2.2) illustrates a list view of articles within a Joomla 4 administration section. The aggregated articles can be reordered and filtered by various filter rules.

Figure 2.2 also illustrates additional basic features. First, the functionality to enable CRUD⁴ actions for a content element, e.g. articles. Second, the feature to bind states (e.g. published, archived, trashed) to content. So, a publishing work-flow can be implemented.

³In [17], the authors address this as *relational content modelling*.

⁴Create, Read, Update, Delete

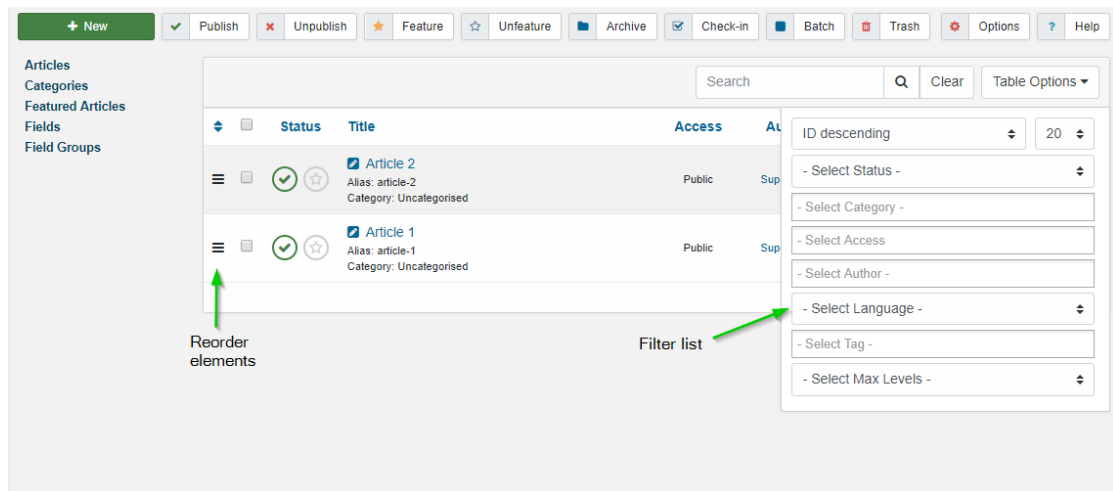


Figure 2.2: Aggregation of Articles as Index Page in Joomla 4 Administration

To enable such features to publishing users of the WCMS instance, a suitable editing interface must be provided by the core system of a WCMS. This interface must support the content owners, editors, and publishers during the publishing work-flow. This includes form elements like *WYSIWYG editors* (allowing rich-text editing), *image and file picker*, *content preview support*, *category selection*, and *publishing options* for content scheduling and expiration. Figure 2.3 illustrates a composition of these features within a WordPress editing page for blog posts within the administration section.

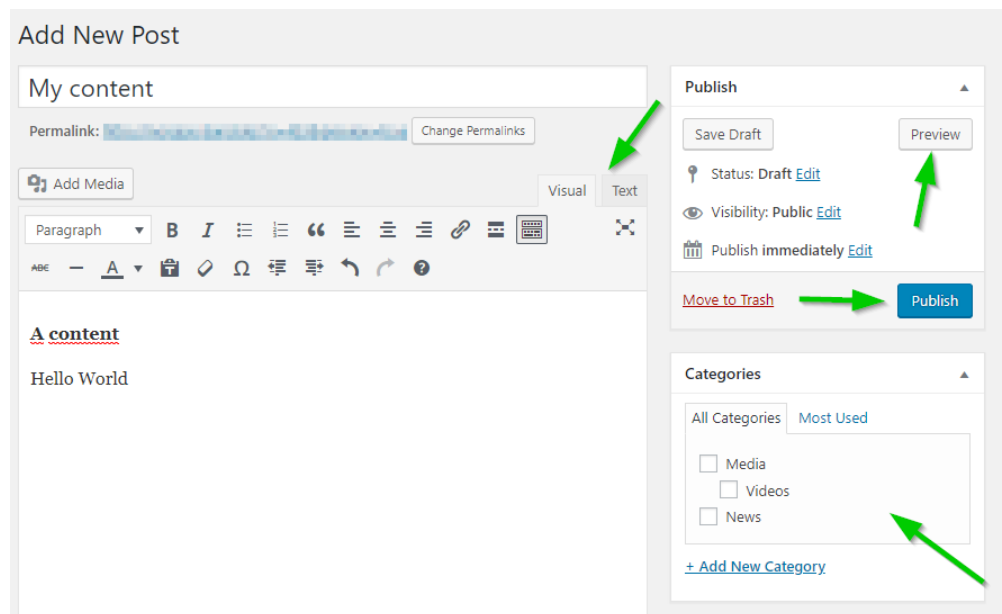


Figure 2.3: Edit Interface for Blog Posts within WordPress 5 Administration Section

Additionally, an editing interface typically comprises a management section for content-related files. This usually includes an option for *file uploads* and *image processing* functionality. Some systems also include *live validation* for the form data, *version management*, and *multi-language associations* as basic functionality.

In order to ensure an appropriate workflow management for various user groups, the core functionality of a WCMS usually provides a configurable permission system. This allows the ACL management for the respective site, i.e. an association of permitted actions based on user roles.

Extended Functionality

In addition to the previously described core functionality, some WCMSs also provide extended features like form building support for content managers or support for multiple languages. Other features like *content search* or *administrator dashboards* are also common as extended functionality in popular WCMSs but are not relevant in the context of this work. Therefore, these features are not further described. A detailed description of them can be found in [17].

By providing *form building* support, WCMSs allow content managers to create individual content structures which are integrated homogeneously to the system. So, individual form fields can be extended by validation rules, error messages, and form data handling (e.g. to send and use form elements as link parameters). Some WCMS provide form building features as part of their core system, whereas other systems depend on third-party extensions for such features. Additionally, systems like Joomla provide mechanisms to extend existing forms by custom form fields [119].

Support for *multiple languages* is another extended feature, which is often provided by WCMSs. If given, a mechanism for managing content in multiple languages can be a powerful feature - especially, if the CMS is used for international presence or used by content managers across national borders. Multi-language support is often provided for various roles. A mechanism for mapping different languages to content and menus is useful for content managers, whereas a mechanism based on language files is useful for all roles including users and content managers, and administrators. The latter mechanism allows an international representation of the system itself. This includes the public section of a running WCMS as well as its administration section. Moreover, such a mechanism is useful for extension developers, since they may use it to enable multi-language support for their extensions. So, once installed to a running WCMS instance, they can be used as homogeneously deployed part of the system.

2.1.3 Extensibility and Programmability

If the core functionality of a WCMS is not sufficient in order to fulfil the functional requirements of a WCMS-based web application, additional features must be implemented. To this end, developers are able to change the platform code of the WCMS. However, a more elegant way is typically enabled by the most WCMSs. Besides the previously described core features, the most WCMSs provide a mechanism for being functionally enhanced by overrides (templating) or deployable software extensions (deeper customization) [17]. So they can be extended without changing the existing platform code. This ensures its further use and smooth updates which should not affect the extension.

Customizations of a running WCMS are typically required by content managers who are often not able to implement them easily due to missing technological knowledge of the system itself. By following the *templating approach*, content managers often create overrides of the core system. Such extensions are typically not affected by system updates, since they are usually bound to templates or themes which represent a special extension kind. So, even content managers without extensive technical knowledge are able to extend a WCMS.

Popular WCMSs, like Joomla, provide features to create and manage core overrides but restrict them to customize views only. So, even though the templating approach allows a quite simple way to customize a running WCMS instance, the dependency to a specific extension kind does not allow a flexible (re-)use of the custom features. Also, deeper customizations are only possible by error-prone workarounds. Therefore, the most WCMSs provide a programming framework, consisting of a flexible API, which enables programmers to develop encapsulated and reusable

software packages - *WCMS extensions*. Typically, these packages can be deployed to a running WCMS instance by an installation dialogue within its administration section. This allows users with the suitable permission (e.g. administrators and content managers) to install and manage extensions via the web application itself (see Figure 2.4). This powerful WCMS feature led to a whole market of WCMS extension in form of extension directories for the most popular WCMSs.

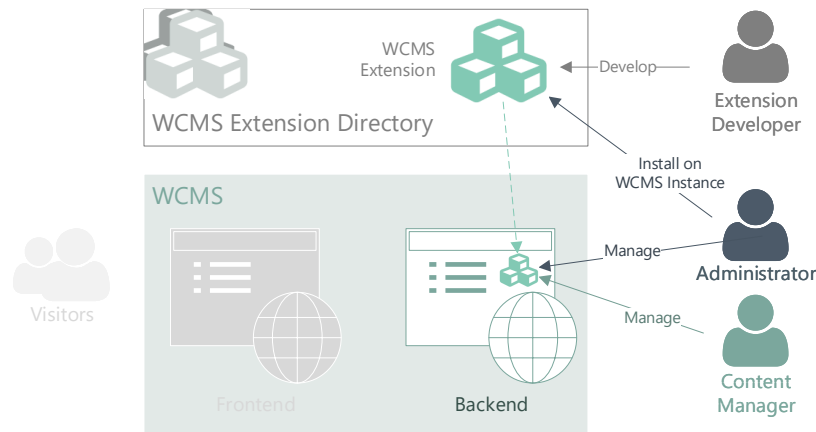


Figure 2.4: Concept of Installable WCMS Extensions

In WCMS extension directories, vendors provide extensions for practically every demand of WCMS users. Although some vendors provide their extensions as part of a commercial package, most of these extensions are available as open source software. The use of such third-party extensions is popular for the most systems, as the number of listed extensions and their downloads shows. The WordPress plugin directory, for instance, features more than 57k plugins with up to 5M downloads of the most popular plugins [266]. The downside of using third-party extensions appears after major platform changes of the WCMS core. Often, missing maintenance leads to version-specific inconsistencies or, in the worst case, that the extension cannot longer be used after a platform update. Due to the fact, that most extensions are small applications by their own, their migration to the new API of the updated WCMS core platform can take months (cf. Section 4.1). Especially the migration of standardized extensions which follow certain code conventions in order to make use a specific WCMS API to ensure homogenous interoperability with the core systems are concerned. Consequently, extension developers avoid the use of a core API or create extensions completely anew, if a new platform version of the respective WCMS is released (cf. Section 4.1).

Depending on the WCMS, various extension kinds exist to customize a running WCMS instance. To customize the appearance of a system, most WCMSs support *templates or themes* as extension kind. This allows installable and reusable packages, which can be used to individualise the style of a WCMS instance. Another extension kind is featured by its *event-based* functional enhancement of a WCMS. So, events like a user login or search request can trigger custom functionality. The APIs of systems like Joomla and WordPress, e.g., enable the implementation of plugins (Joomla) and actions (WordPress) for this purpose.

An additional popular extension kind typically encloses a set of *segregated but related features* with a separated configuration part and often an own database part, once installed to a WCMS instance. Usually, these extensions can represent complete applications within a running WCMS. Examples for such extension types are shop systems, calendars, media and file managers, backup managers, or form builders. The variety of features is nearly limitless and almost all necessi-

ties of content managers are covered by existing extensions of this type. In WordPress, this extension kind is represented by plugins, whereas Joomla calls them components, and Drupal modules. Often, these extensions are augmented by a related but different extension kind for the *representation of existing data* within a certain template position on a WCMS instance. These extensions are often called widgets or, as in Joomla, modules.

Besides the previously described extension types, *libraries*, *language file plugins*, and *editor plugins* for the customization of content editors are commonly used.

Typically, interdependencies between extensions exist. As previously mentioned, these dependencies happen between extensions that manage and extensions that represent the same data (e.g. Joomla components and modules). Additionally, dependencies between these extensions and language plugins and libraries are common and often inevitable. The fact that extensions may depend on existing data or features of other extensions has to be considered during their further development and maintenance.

Some systems like Joomla and Drupal also use extensions for the encapsulation of their core features. Joomla, for example, uses components for their basic features like user, content, menu, and media management, whereas it provides several representation modules as well as search, authentication, and system plugins. This allows administrators to enable/disable core features to downsize a running instance to their specific needs and hide irrelevant features from content managers.

2.2 Commonalities and Differences in WCMS Extension Development

Various popular WCMSs exist in the WCMS market. Most of these systems provide the previously described features and can be customized by functional software extensions. In order to find a suitable reference WCMS for the research of this work, we explore the WCMS market and compare the most popular and suitable WCMSs by their extension features.

For an objective comparison, we adhere to the following criteria which influence the extensibility of the respective WCMS: popularity, community, version strategy, core feature integration (content management, multi-site management, language handling, file handling, menu management or page composition, user management and ACL, performance), and extensibility features (complexity, architecture, API, backwards-compatibility). So, during the following comparison, we take the previously presented WCMS features into account and consider the influential aspects for WCMS extension development. The result of the comparison will reveal a suitable reference WCMS for the research scope of this work.

2.2.1 WCMS Market

The most popular WCMSs, or so called “second-generation content management systems” [160], which support their extensibility through installable software extensions are WordPress [266], Joomla [170], and Drupal [33].

Other well-known CMSs like Magento [144], Shopify [212], and Blogger [80] are even widely spread, but the market share is significantly lower compared to the previously mentioned top 3 WCMSs. Additionally, their functionality is mainly tailored to one specific domain. For instance, Magento and Shopify are used for web shop instances, whereas Blogger is intended exclusively for web blogs. However, we must additionally mention TYPO3 [239] here, since it is one of the most matured WCMSs and enjoyed much popularity in industry throughout the last decade. Other systems like Wix [259] or Squarespace [219] enjoy high popularity in media due to their design-centric features. However, such systems are typically used as site builders and cannot be

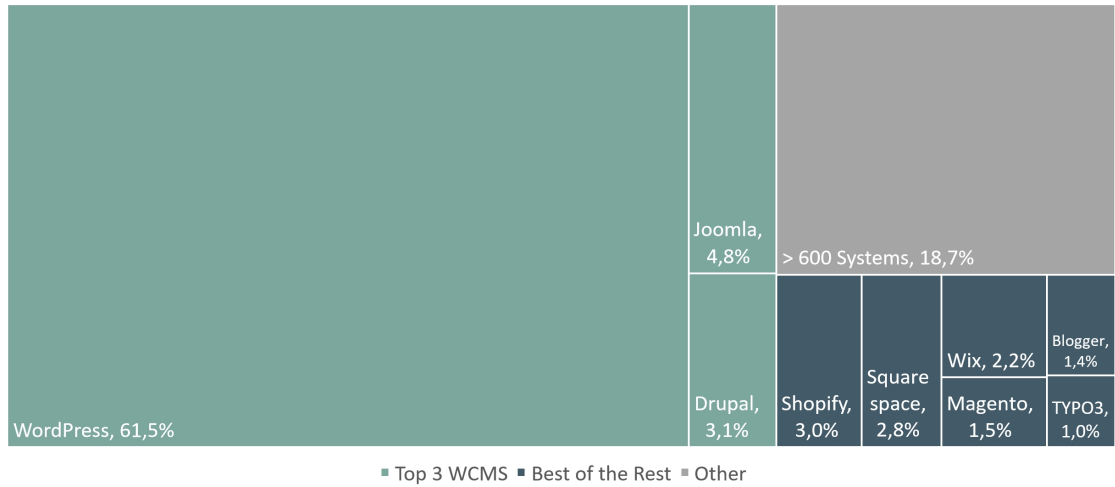


Figure 2.5: WCMS market share (based on [252], stand October 2019)

customized in the sophisticated way as the top 3 WCMS. Another WCMS to mention is Contentful, which acquired popularity due to its modular integration of various services. WCMSs like Contentful and Strapi [223] are characterized by their decoupling of content and its representation. Therefore they are also known as *headless CMSs*. Such features can also be enabled by traditional WCMSs by web service implementations. One of the most popular services using Contentful is Spotify.

Within the following subsections we present the most popular WCMSs and illustrate their similarities and differences in order to find a first suitable reference WCMS for an MDE approach. We will only select systems, which are flexible enough to allow all kinds of web applications without restriction to one specific feature. I.e. we exclude page builders like Wix or Squarespace, mere blogging systems like blogger, and feature-specific systems like Magento and Shopify. Even though the Contentful WCMS and other headless CMSs like Strapi gained popularity over the last years, their popularity is still small compared to the top 3 WCMSs. Additionally, their architecture does not allow general conclusions. Therefore, they cannot be used as reference WCMS and will not be considered during the following comparison.

WordPress

Over the last decade WordPress became the most popular open source CMS in the web domain. Currently, around 80 million⁵ websites are based on WordPress. This corresponds to a WCMS market share of 61.5% (cf. Figure 2.5). Although WordPress was intended as weblog software, it evolved as full WCMS, since it exhibits much of the same functionality as other WCMSs. This includes editorial functions for content management, role based user management, page composition, administration interface. According to [264], WordPress is distinguished by its ease of installation, which takes no longer than 5 minutes.

Since early versions of WordPress, its core platform enables developers to customize and extend its base functionality through different extensions kinds. Themes can be implemented to enclose style definitions as reusable packages, whereas plugins allow the functional extension of a WordPress instance. Existing themes and plugins are provided via an extension directory which can directly be accessed from the administration section of a WordPress instance. In Section 2.2.2, we present the WordPress extension mechanism in more detail.

⁵Stand June 2019. According to the total amount of 235 million unique and active domains [159] and a share of 34% of WordPress [252].

Like other Open Source WCMSs, WordPress has a large and active community, even though WordPress itself is profit-oriented (e.g. by using the WordPress hosting platform `wordpress.com`). So, support and documentation for installation, operation, and extension development is provided to WordPress administrators and content managers.

During its life cycle, WordPress was subject to several platform improvements in form of new major version releases. According to [265], WordPress underwent 34 main releases in 5 major versions since version 1 in 2004. With every major version, the underlying platform evolution required extension developers to migrate their extensions to the new WordPress core in order to ensure their functional operability on deployed and migrated WordPress instances.

Joomla

Like WordPress, the open source WCMS Joomla gained high popularity during the last decade. However, the market share is significant lower, which is due to the fact that the Joomla system possesses a higher complexity than WordPress. Nowadays, around 6.5 million websites are based on Joomla according to actual web statistics ([252] and [159]).

Even though Joomla requires more technical affinity in contrast to WordPress the higher complexity allows the creation of more flexible and dynamic web applications, whereas WordPress is mostly used as basis for simple websites with a high amount of static content. Analogous to WordPress, Joomla offers a rich range of functionality to enable the creation of dynamic content-based web applications. This also includes content and menu management, an administration interface, multi-language handling, and flexible ACL features. However, the biggest strength of Joomla is its sophisticated extension mechanism which will be explained in detail below (see Section 2.2.2).

Joomla is a community-driven project. Therefore, many contact points exist which support administrators, content managers, and extension developers. This includes forums and documentation platforms maintained by the community itself. Another contact point is the extension directory which introduces existing Joomla extensions. This directory is also available from the administration section of a Joomla instance, similar to WordPress.

Since the first version release in 2005, 16 main and 3 more major releases were published by the Joomla community. In the past, the version strategy of Joomla was considerably flexible. With the 1.5 version release in 2008 the extension API became more sophisticated, e.g. by requiring the implementation of a model-view-controller pattern in Joomla components. This pattern has established itself through the simplification and standardisation during development of own extensions. The next major changes came with the release of version 1.6 and 1.7. which extended this API by new ACL functionality. In order to follow a more strict versioning strategy, the major changes of 1.6 and 1.7 flew directly into the long-term-support version 2.5. Since version 3, Joomla follows a strict version strategy by adhering to the semantic versioning guidelines [182]. The current version 4 (beta) release comes with completely new features, such as a publishing workflow, web service integration, and a new extension API.

As the version history of Joomla shows, major version changes came with tremendous architectural changes of the system. Hence, existing third-party extensions were immediately influenced and had to be migrated to new system API requirements.

Drupal

Drupal is one of the most widely used WCMSs besides WordPress and Joomla. Based on [252], its popularity established directly behind Joomla with a market share of 3.1% (cf. Figure 2.5). This means, according to the presented statistics in [252] and [159], that in 2019 4.6 million websites are based on Drupal.

From a structural and technical perspective, Drupal does not differ tremendously from the other presented WCMSs. It comes with similar core functionality for content and user management, ACL support, multi-language handling, and is extensible by functional extensions which are called modules. In addition, Drupal stands out for its multi-site capability. However, unlike WordPress or Joomla, Drupal is intended to be used as community platform. To this end, various social-community extensions are provided to enable blog features, forums, and social media integration. Therefore, most of the core features have to be added manually after the installation of a Drupal instance to keep the system lean. This feature can be seen as a drawback, if Drupal shall be used as functional rich and full WCMS similar to WordPress and Joomla. Though, the Drupal community provides distributions which enclose the Drupal core and suitable modules for various purposes.

Another feature, what can be seen as drawback by some Drupal users, is its administration interface. Systems like WordPress and Joomla provide a separate administration backend, whereas Drupal has integrated the administration into the frontend of the system. This means, no specific interface is provided - the management functionality can be accessed by sub menus depending on the access rights of the user. Typically, such a feature is also provided by the other top WCMSs. Though, a common workaround is the use of a configuration theme, which enables a nearly separated administration interface.

One of the biggest strengths of Drupal is its community support. Similar to the Joomla project, forums, mailing lists and documentation sites exist, which are driven by the community. Additionally, an extensive extension directory exists. Like the Joomla core, the core system of Drupal is implemented by developers of the community. Therefore, contribution documentation for developers is also available in many languages.

As mentioned before, Drupal can be customized by functional extensions. Similar to other systems, the maintenance of the extensions depends directly on every update of the system core. Drupal evinced 9 major versions since its first release in 2001. Similar to other systems, the releases typically came with tremendous API changes, which directly affected all installed modules on a running Drupal instance. Moreover, even minor releases sometimes brought changes to the core, which affected the operability of installed modules. Therefore, all installed modules must be checked and re-implemented to guarantee flawless Drupal instances. In contrast to WordPress and Joomla, this also includes modules with typical core functionality. Especially modules with dependencies to other modules challenge site administrators. If a dependency is not migrated to the new version, the dependent module usually cannot be used as well. According to [10], Drupal instances typically have 50-60 module dependencies. Thus, this missing backwards compatibility is one of the biggest disadvantages of Drupal.

2.2.2 Extension Development Comparison

Based upon similar popularity, core features, and extensibility, we can assume, that the previously introduced top 3 general⁶ WCMSs (WordPress, Joomla, and Drupal) will remain in a healthy state of competition for a longer period of time. Each of these systems has a big community standing behind it and is continuously enhanced by contributors around the globe.

In order to *determine a suitable reference system for the profitability research of a model-driven approach during extension development*, we compare the extensibility features of these systems under consideration of the prevailing extensibility features which are described in Section 2.1.3. Related work in this context, provide comparisons of overall features but not for extension development (cf. [148]) or only consider security aspects (cf. [151]).

⁶As of September 2020, Shopify has overtaken Joomla and Drupal in the WCMS usage statistics [252]. Due to its limitation for WCMS-based web shop systems, it is not further considered in this work.

We follow the straightforward Software Architecture Comparison Analysis Method (SACAM) as proposed in [222]:

1. *Preparation*: Identifies the relevant business goals needed in the comparison and examines available documentation for each architecture candidate.
2. *Criteria Collation*: Derives comparison criteria from the business goals and refines them to quality attribute scenarios.
3. *Determination of Extraction Directives*: Determines the architectural views, tactics, styles, and patterns that are looked for during the following extractions to find supporting evidence for the scenarios of Step 2.
4. *View and Indicator Extraction*: Extracts the architectural views for each candidate according to the extraction directives from step 3. Detects indicators that support the quality attribute scenarios from Step 2. Architecture recovery techniques may be needed to generate relevant views.
5. *Scoring*: Scores the fitness of a candidate architecture to support the criteria.
6. *Summary*: Summarizes the analysis results and provides a recommendation for the decision-making process.

For each top 3 WCMS, we *explore the complexity and architecture of the respective extension development API and assess developer support during extension development by documentation and the community*. This also includes the *determination of challenges during development and inspection of existing development infrastructures* which support extension developers and quality assurance support, e.g. by test suites and coding standards. So we can *ensure support during the further elicitation of a model-driven engineering infrastructure* for WCMS extensions. Additionally, we *explore the WCMS-specific extension kinds and their features* like core support, database support, section-specific views (frontend/backend), homogeneous integration to a running WCMS instance, and interoperability with other extensions.

After a separate analysis of the extensibility of the top 3 WCMSs, we present a direct comparison of the extensibility features. This includes a quantitative scoring for *developer support, extension features* and the *API* of each system. So, we aim to *identify the system with the most sophisticated extension mechanism* in an impartial manner.

During the comparison, we will not consider strategies for local development environments using e.g. XAMPP, MAMP, LAMP or WAMP stacks in detail. Tools like Bitnami for XAMPP [11] support most of the mentioned WCMSs, including WordPress, Joomla, and Drupal during local extension development equally. Therefore, a WCMS-specific comparison of this feature is not mandatory.

Development of WordPress Extensions

As mentioned before, WordPress allows developers to extend the core functionality by functional *plugins* and style definitions in form of installable *themes*. The community supports extension developers in forums [262] and by an extensive documentation [263]. The latter includes detailed guidelines on how to develop WordPress plugins. Even a documentation for security assurance is available, since WordPress websites are popular targets of hacker attacks. Additional developer support is given by existing boilerplate generators, even though these are not provided by the WordPress project itself.

WordPress does not store any extension-specific information in the database which allows a straightforward plugin deployment during development. It directly scans a specific folder on the file system and integrates all contained plugins.

Besides plugins and themes no other extension kinds are provided. So, all extension features are directly covered by these two kinds. Themes are used for the reusable implementation of style definitions, whereas plugins incorporate all possible features for the functional extension of the WordPress core system. These features include core support (e.g. user access), own administration menu definition, short-code support for content editors, JavaScript support (Ajax and jQuery), cronjob integration, and internationalization and localization support, provided by a sophisticated multi-language translation mechanism.

WordPress provides an API for plugins and themes, even though the use of it is not mandatory. This API is also used by the core itself, since it offers functions for database, core, and webservice features. However, it is also possible to implement own functions without using the API. The complexity of WordPress plugins can vary from simple plugins for content filtering up to extensive integrated applications like a web shop. Since there is *no development standard* for WordPress plugins, developers are free to decide if they implement them based on a structured architecture or put everything into one PHP file. The latter, *a main PHP file, is the only artefact WordPress requires*. This file must consist of a header DocBlock in a given format with the plugin name as minimum specification (Listing 2.1). Additionally, plugin information like description, version, and author information can be placed in this DocBlock. Another configuration file is not needed.

```
1 <?php
2 /**
3  * Plugin Name: MyWordPressPlugin
4  */
```

Listing 2.1: Minimum specification within the main PHP file of a WordPress plugin

In addition to the configuration DocBlock, the main PHP file may contain implementations for additional features in form of PHP functions which can be assigned to WordPress events as parameters (*hooks*). So, the *implementation of event-handlers* is required, which are triggered by specific events calling a given function (callback).

```
1 function my_action_function() {
2     // custom actions
3 }
4 add_action( 'init', 'my_function' );
5
6 function my_filter_function() {
7     // custom filter
8 }
9 add_filter( 'the_title', 'my_filter_function' );
```

Listing 2.2: WordPress action hook

This programmatic feature is similar to other event-based implementations like e.g. event handlers in JavaScript. WordPress provides two kinds of hooks: actions and filters. Action hooks include system events like initialisation, (de-)installation, or user registration/login/logout, whereas filter hooks aggregate content-centric events.

Listing 2.2 presents the required structure of an action hook which has a custom function (`my_action_function()`) given as callback for the `init` event and a filter hook which calls the `my_filter_function()` when the `the_title` event is triggered. WordPress also allows to register custom hooks which can be used within other plugins.

Since WordPress plugins mainly consist of PHP functions, *developers have to avoid name conflicts between their and other installed plugins*. The WordPress documentation recommends the straightforward use of prefixes or the implementation of OO by own classes. Based on this chal-

lenge and given the fact that everything is incorporated in one php file (plugin manifest, PHP code for functions and hooks, HTML and CSS definitions, and often JavaScript code), some *architectural patterns have become popular best practices* in the WCMS community. Even though these patterns are not directly required by the core system, extension developers can increase the internal code quality tremendously by implementing them, following the *separation of concerns* design. One recommended pattern, *Slash*, incorporates singletons, loaders, actions, screens, and handlers [101], whereas another popular pattern is based on a *model-view-controller (MVC)* pattern. Both patterns are also applicable for themes development, since the mechanism is rather similar. In order to separate administration functionality from public features and translation rules, a minimum file structure for plugins is recommended by the community.

In conclusion, developers are free to implement plugins in their preferred way. This allows a straightforward deployment to a running WordPress instance, since the only requirement is a configuration header in a PHP file. No other files, e.g. for creating database tables during installation are required. However, such features must be considered by plugin developers and are not validated by the system itself. This can be seen as an disadvantage, since the main responsibility for a homogenous interplay between the system and the extension must be guaranteed by the developers. Another drawback of the philosophy appears, if an existing plugin has to be refactored, e.g. by different developers. Even experienced developers may require additional effort to understand the code of a third-party plugin, since there is no given standard structure.

Development of Joomla Extensions

In contrast to the extension mechanism of WordPress, Joomla supports a set of different extension kinds. So, Joomla separates extension kinds based on their complexity and function, whereas WordPress only provides two extension kinds which can include features of various complexity. Similar to WordPress, an extension kind (*templates*) for the reusable style definition of a running instance exists. Joomla *plugins* represent an extension kind for the implementation of actions which are initiated by specific events (e.g. authentication, search, or content loading). These plugins are comparable to WordPress plugins, whereas more sophisticated feature implementations should be placed in provided extension kinds: *components* and *modules*. Components represent mini applications, usually coming with their own database tables, management views for the administration section (backend), and views for the public section (frontend) of a Joomla instance. Modules encapsulate lightweight features for placing any information, mostly data of one or more components, on specific template positions on a Joomla instance, such as widgets. A *package* is a special extension type, since it allows to group extensions into one installable extension. So, several extensions can be installed during one installation routine.

Similar to WordPress, developers of Joomla extensions are also supported by an extensive documentation [105], including tutorials and API description. This documentation is also tailored to the Joomla-specific extension kinds. However, components get the most attention, since they represent the most complex extension kind. The developer community of Joomla also provides support in forums and developer workshops which were given during national and international community conferences. To ensure high quality extensions, the community proposes an own coding standard and alternatively recommends to follow the PSR-12 standard recommendation for PHP [179]. In addition, a test suite for extensions is also provided by the Joomla project.

To speed up extension development, a variety of boilerplate generators exist. Some of them, like the Joomla component builder [242] can be used as Joomla extension from the administration section of a Joomla instance. Others are available as IDE plugins, like the Joomla boilerplate generator which is integrated in JetBrains's PhpStorm IDE, or are available online, like the Joomla component generator [217] or the component creator [102]. The limitation of these boilerplate generators is the *missing or limited support for other extension kinds besides components and interoperability between various extensions*.

The sophisticated extension mechanism of Joomla allows extensions to be homogeneously integrated to running Joomla instances. For each extension kind, a specific administration section is provided. So, administrators can restrict extension configuration explicitly for specific user roles. An example is the creation and configuration of a module instance in the backend, which could be accessible for content managers, whereas the plugin configuration section is restricted exclusively to administrators. In contrast to WordPress, such restrictions must not be considered by extension developers. Independent to the extension kind, developers can make use of Joomla's translation mechanism (internationalization and localization) which is based on language files with key-value definition and language keys as output values. It is also possible to implement language packages, which can be installed to a running Joomla instance. During the installation routine of a new Joomla instance, administrators can directly download and configure existing language packages from the community.

Similar to other WCMSs, Joomla provides an API, which provides simplified functions for common PHP features, enables core support (e.g. access to the user object), and offers an interface for database operations. The API is also used by core extensions which implement the whole functionality of the core itself. Moreover, the API can also be used as independent PHP framework⁷ To ensure a homogeneous integration, *Joomla extensions should implement architectural patterns, required by the Joomla core*. To this end, Joomla makes use of some popular design patterns of software engineering which will be elaborated on below. Based on the extension kind, different patterns are intended to be used.

As stated before, plugins for Joomla follow a similar event-based architecture like WordPress plugins. However, WordPress requires the registration of event handlers, whereas Joomla forces extension developers to *implement the observer pattern*. More specific, developers have to implement PHP observer classes which extend the API's abstract observer plugin class. Additionally, they have to *follow a specific name pattern* including the plugin type, e.g. authentication, content, system, or user. Based on the plugin type, associated methods can be implemented which are invoked automatically if a specific event occurs. Listing 2.3 shows an excerpt of the class structure of the Joomla user plugin which implements methods which are triggered by user-specific events - in this case the login and logout of a user.

```

1 class PlgUserJoomla extends CMSPlugin
2 {
3     public function onUserLogin($user , $options = array())
4     {
5         // Check, if user is blocked, create session, ...
6     }
7     public function onUserLogout($user , $options = array())
8     {...}
9 }
```

Listing 2.3: Joomla plugin structure (Joomla's user plugin)

Content plugins are used to modify content before it gets prepared for being displayed. This enables short-code support for content editors. So, custom placeholders can be specified which can be used during content creation and are replaced by content plugins. This powerful feature allows dynamic content adaption.

In contrast to WordPress, the Joomla core is intended to be extensively augmented by functional extensions. Plugins are sufficient for event-based functionality but are not designated for the implementation of an additional application within a Joomla-based site. Therefore, Joomla provides an architectural interface for such functionality for components, while in WordPress, extension developers have to create their own architecture within the plugin environment. To this

⁷The Joomla! Framework project [106], see the official project documentation in [167].

end, *another popular design pattern is implemented in the context of components - MVC*. Joomla uses this pattern to separate the presentation, functionality and data handling of components. To make use of the core API, component *developers have to implement this pattern in a required structure on code and file level*. Moreover, by following this standard, the interoperability between the core system and installed components can be guaranteed. If developers do not adhere to this pattern, error-prone extensions are highly probable. It is possible to create components in a different way (e.g. by putting all the functionality to one PHP file). However, this is uncommon in the domain and developers who implement such extensions are working against the community standards.

Even though the architectural structure of all extension kinds is different, they have some common elements. The first common part of all Joomla extensions is represented by a configuration file called *manifest*. Independent to the specific extension kind, extensions must consist of one manifest file [107]. This XML file is required during the installation procedure of an extension. It consists of extension metadata details such as name, extension kind, description, author, copyright, and license. Additionally, developers have to specify the file structure of their installation packages. In addition, the path to the component-specific language files can be specified in the manifest file. Dependent on the extension kind, the files of the extension package are moved to specific sub-folders of the Joomla instance during the installation routine. Components, for instance, usually provide files for the backend and the frontend which should be separated in the installation package. However, Joomla does not require a strict file structure for installation packages. During installation, the backend files are moved to a respective extension folder `<Rootfolder>\administrator\components\<nameofthecomponent>`, whereas all frontend files will be copied to the `<Rootfolder>\components\<nameofthecomponent>` folder. The same applies to language and media files which are moved to respective folders of the Joomla instance. This approach differs from WordPress, where all plugin files are placed within one plugin folder, and shows the relevance of correct definitions in the manifest files of a Joomla extension. Listing 2.4 shows an excerpt of the manifest file of Joomla's core user component.

```

1 <extension type="component" version="4.0" method="upgrade">
2   <name>com_users</name>
3   <author>Joomla! Project</author>
4   <version>4.0.0</version>
5   ...
6   <namespace>Joomla\Component\Users</namespace>
7   <files folder="site">
8     ...
9   </files>
10  <languages folder="site">
11    <language tag="en-GB">language/en-GB.com_users.ini</language>
12  </languages>
13  ...
14  <administration>
15    <files folder="admin">
16      <filename>users.xml</filename>
17      <folder>Controller</folder>
18      ...
19    </files>
20    <languages folder="admin">
21      ...
22    </languages>
23  </administration>
24 </extension>

```

Listing 2.4: Joomla manifest structure (Joomla's user component)

The implementation of a required design pattern and standardized configuration structures may ensure maintainable and reusable extensions. However, this approach also has some drawbacks. The flexibility for extension developers is not given, since *every discrepancy could lead to error-prone extensions*. In addition, the required structure necessitates strict name conventions on code and file level. This *could lead to errors which may occur during runtime*, due to the fact that PHP is used as interpreter language. This can be a hurdle for new developers in the domain, but also for experienced developers who have to maintain existing extensions.

To sum up, extensibility is a main driver in the Joomla community. Therefore, a strong focus is placed on the interoperability between extensions and the core system. To this end, Joomla requires the implementation of popular design patterns to ensure homogeneous integration of extensions. So, developers can focus on their actual feature implementation without the need of considering, e.g., permission features. Additionally, *a standardized extension structure enables developers to refactor and re-engineer legacy extensions of other developers*. The support of various extension kinds allows extension developers to implement extensions in a more flexible and reusable way. Moreover, this separation allows the core system to provide features for the specific extension kinds, like the individual management of them in the administration section of a Joomla instance. A drawback of Joomla's extension mechanism is the *missing standard file structure* for installation packages, even though some recommendations exist. Developers are in charge to map the file structure to the given structure of the Joomla core system where the files will be moved during the installation routine. This complicates the deployment during development, since extensions cannot directly be copied to a certain folder as it is common in WordPress. Another challenge which relates to this fact is the extraction from installed extensions to create installable extension packages. In WordPress it is sufficient to extract the plugin folder which contains all the plugin files, whereas in Joomla the extension files must be extracted from different directories.

Development of Drupal Extensions

Drupal offers an extension mechanism which is comparable to the flexible but not standardized WordPress mechanism based on a few provided extension kinds. However, similar to Joomla, Drupal *requires compliance with certain coding standards and architectural patterns*. These *standards* are well described in a lucid documentation including coding, UI, and documentation standards. Additionally, the documentation supports extension developers with guides for local development, testing, development tools, and security assurance.

In addition to the documentation, Drupal developers are supported by a variety of boilerplate generators and IDE integrations. The latter includes popular IDEs like Eclipse, Aptana, Netbeans and PhpStorm. Similar to Joomla, installable extensions exist, which provide the development of Drupal modules within a running Drupal instance. So, developers are able to develop an extension in the Drupal environment without the need of any IDE. Like WordPress, Drupal also provides cronjob support which allows automated actions, even for features of individual extensions. In order to support quality assurance, Drupal recommends the use of a provided PHP Unit test environment [57]. A special feature, which is also provided by Joomla, is the provided dependency management support (using Composer [1]).

The provided extension kinds of Drupal include *themes* for reusable style definitions (like themes in WordPress and templates in Joomla), and *modules* for functional features. The scope of functions of a module can be as complex as desired, what is similar to WordPress. Modules are classified as *core* (modules are shipped with Drupal and were developed by developers of the Drupal core and the community), *contributed* (third-party - own developments of the community and free of charge), and *custom* (custom developments for specific problem solutions of single Drupal pages). Similar to Joomla, the functionality of the core system is also encapsulated within modules. In addition to themes and modules, Drupal also allows bundles of reusable features within

library-like extensions called *plugins*. These should not be mixed-up with WordPress or Joomla plugins, which use the naming differently. A special extension kind in Drupal is represented by installation profiles, which pack various modules and plugins together for specific needs. So, during installation, administrators can narrow down the functional scope of their Drupal instance more detailed and tailored to their specific needs in order to acquire lean applications.

To enable interoperability between extensions, Drupal extension developers can make use of existing core, contributed, or custom modules in their modules. Thus, core features for content manipulation or user interactions can be integrated into custom modules. To ensure own administration interfaces, menus can be specified, even though *an actual separation into a backend and frontend is not implemented in Drupal*. Similar to WordPress, Drupal allows to specify cronjobs, even for custom modules. For internationalization and localization support, Drupal provides a simple translation mechanism based on language files.

Similar to WordPress, Drupal extensions are automatically discovered, even though Drupal also provides extension deployment through an installation routine in the application's dashboard. It is sufficient to place the extension files in specific folders of the instance, e.g. `<Rootfolder>\modules\custom\<nameofth module>`. For a proper interoperability with the Drupal instance, a main configuration file is needed, which has to follow a certain name pattern (`<name of the module>.info.yml`) and must include some basic parameters. The example configuration for a `helloWorld` module [56] in Listing 2.5 illustrates a configuration file with all possible configuration parameters.

```
1 name: Hello World Module
2 description: Creates a page showing "Hello World".
3 package: Custom
4
5 type: module
6 core: 8.x
7
8 dependencies:
9   - drupal:link
10  - drupal:views
11  - paragraphs:paragraphs
12  - webform:webform (>=8.x-5.x)
13
14 configure: hello_world.settings
15
16 php: 5.6
```

Listing 2.5: Drupal configuration file (helloWorld example)

Drupal requires **name**, **type**, and **core** as basic properties. To group extension within the administration representation of a Drupal instance, it is recommended to use the **package** parameter (cf. Figure 2.6). So, Drupal provides a categorisation view for extensions, which allows the specification of own category names.

Special emphasis should be placed on the **dependencies** parameter. This parameter can be used for the specification of extensions, which must be installed on the host instance. Otherwise, dependencies to these extensions cannot be solved, what could lead to errors when the custom extension is executed. Such a feature is also implemented in other WCMSs such as Joomla (e.g. if modules depend on existing components). By specifying dependencies in the main configuration file ensures a message in the administration view, if dependencies are missing. If the site administrator uses Composer as dependency manager, extensions to which dependencies exist can be installed automatically. This, however, requires an additional specification in a composer file.

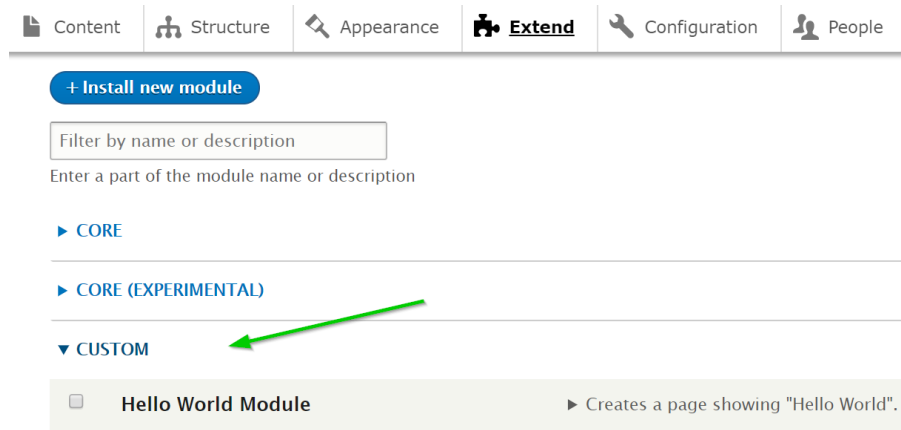


Figure 2.6: Module representation within a running Drupal instance (administrator view)

Like WordPress and Joomla, Drupal also provides an API as interface for common PHP features and database access. The Drupal API is built on the PHP framework Symfony [210], *forcing extension developers to implement their extensions object-oriented*. Additionally, the API makes use of the factory pattern and late static bindings, a PHP feature which can be used to reference a called class in the context of static inheritance. Also, the Drupal API allows extension developers to make extensive use of dependency injection and event-based features by hooks (similar to WordPress). Another speciality of the Drupal API, based on the Symfony framework, is given by the support of the template engine TWIG. This template engine can be used for a proper integration of HTML and PHP code, increasing the internal quality of the code. However, *the Drupal API does not support other popular design patterns like MVC*. Nevertheless, it should be used by extension developers to ensure a homogeneous integration of e.g. modules into a running Drupal instance. To this end, Drupal follows a *configuration-based approach using YAML configuration files* for extension metadata. Additionally, since Drupal 8 other features are also configured like routing mappings or menu links, whereas up to Drupal 7, exclusively hooks (event-handlers with callback functions) were used. Provided that the Drupal API is used in the expected manner based on Drupal's coding conventions the API supports a convention over configuration paradigm, keeping the configuration files as basic as possible.

Summarizing, the Drupal community attaches great importance on developer support in order to ensure standardized extensions of high internal quality. To this end, an extensive documentation is provided which offers detailed information regarding coding standards, local development, testing, development tools, and security assurance. In addition, vast IDE support and boilerplate generators are available. Besides an extension kind for appearance configuration (themes), Drupal allows the installation of functional extensions (modules and plugins). Similar to Joomla, Drupal also uses these extension kinds for the realisation of its core functionality. This enables a straightforward interoperability between custom and core modules. Additional extension features are: own administration interfaces, cronjob integration, and internationalization and localization support based on language files. The Drupal API, which is based on the Symfony framework, provides common PHP features and database access. Based on the API, extension developer have to make use of OO. A special feature of the Drupal API is the support of dependency injection and hooks for the registration of custom actions based on Drupal events. Additionally, the template engine TWIG can be used for an elegant integration of HTML and PHP code. Popular design patterns like MVC are not supported directly by the Drupal API. The Drupal core supports YAML configuration files for extension configuration, routing mappings, and menu links. Thus, Drupal follows a convention over configuration approach which allows basic configuration files, provided that the API is used as expected within the extension implementation.

Overall Comparison of Extensibility Features

In this section, we compare the extensibility of the previously considered systems in order to elicit a suitable reference WCMS as basis for our research of applying an MDE approach in the WCMS domain. To this end, we contrast the presented extensibility features and identify the system with the most powerful and complex extension mechanism. So, we can follow a bottom-up approach by using this system as reference WCMS and transfer the findings to other WCMSs with an equal or less sophisticated extension mechanism.

To evaluate the selected features, we use the following scheme:

- ✗: Feature not provided
- (✓): Feature provided to some extend (rudimentary or workaround)
- ✓: Feature provided
- ✓✓: Feature provided (sophisticated)

The first comparison criteria considers the developer support provided by the respective community. This includes existing documentation, active forums, coding standards and existing test suites. These artefacts are supportive during the development of new suitable tools for applying model-driven development in the domain.

Table 2.1: Developer Support

WCMS	Forums	Documentation	Coding Standard	Test Suite
WordPress	✓	✓	✗	✗
Joomla	✓	✓	✓	✓
Drupal	✓	✓	✓	✓

All of the presented systems provide forums and helpful documentation, whereas Joomla and Drupal also provide community-valid coding standards and a test suite for quality assurance. Even though test infrastructures for WordPress exist, these are not directly provided by the community itself.

Another comparison aspect considers the characteristics of WCMS extensions. This includes the range of extension kinds, interoperability support between extensions and WCMS core, supported features like integrated internationalization and localization, dependency management, and Cronjob support. These aspects are the main requirements in order to evaluate existing or new developed model-driven development tools.

Table 2.2: Extension Features (1)

WCMS	Template/ Theme type	Separation of functional extension types	Interoperability between extensions	Core Support
WordPress	✓	✗	✓	✓
Joomla	✓	✓✓	✓	✓
Drupal	✓	(✓)	✓	✓

All systems provide an extension type for reusable style definitions which can be used for the configuration of the appearance of the respective WCMS instance. Additionally, extension kinds for functional augmentation are supported by the core of all presented systems. As stated before,

a variety of names for several extension kinds exist. Some of these names are intuitively chosen like libraries or templates/themes, whereas others are used for different purposes depending on the respective WCMS. However, WordPress and Drupal provide one main extension kind for functional extension, whereas Joomla stands out for its sophisticated separation into various extension types. This allows extension developers to encapsulate their features into suitable packages based on the respective complexity (mini application, website widget, event-based actions). Drupal allows the encapsulation of re-usable interfaces into an additional extension type, whereby WordPress developers can incorporate features of any complexity within one extension type. With respect to maintenance, standardization, and re-usability, Joomla's approach is significant better.

Additionally, the presented systems allow interoperability between extensions. A powerful feature which allows developers to (re-)use functionality from other extensions within their own extensions. Since Joomla and Drupal also use their extension mechanism for the implementation of their core features, the provided interoperability also enables core support. WordPress provides core support as well. However, extension developers must use the WordPress API in order to use core features in their extensions.

Table 2.3: Extension Features (2)

WCMS	Internationalization/ Localization	Dependency management	Cronjob support
WordPress	✓✓	✗	✓
Joomla	✓	✓	✗
Drupal	(✓)	✓	✓

The provided extension features of the selected WCMSs comprise internationalisation and localization features based on translation files. WordPress, however, stands out for its sophisticated translation mechanism based on reusable meta language files. The mechanism provided by Drupal is rather basic and does not support full localization support. In addition to this feature, Joomla and Drupal extensions can include automated dependency management to ensure proper installations within a running website. Another feature, cronjob integration, is supported by WordPress and Drupal. Joomla lacks such a feature innately.

Considering the API support for extension development, we compare supported development conventions and implemented design patterns. Additionally, we compare the file and code structure of installable extension packages and their capability of homogenous integration into a running WCMS instance. These aspects play a significant role during the elicitation of a suitable domain-specific modelling language which has to generalize them properly.

Table 2.4: API Support

WCMS	OOP	Architectural patterns (e.g MVC)	Event-based support
WordPress	(✓)	✗	✓
Joomla	✓	✓	✓
Drupal	✓	✗	✓

The Joomla and Drupal APIs require developers to implement their extensions in an object-oriented manner, whereas the WordPress API does not require an OO implementation. Though, WordPress supports all kinds of implementations. The WordPress and Drupal APIs do not support architectural patterns directly, even though some recommendations exist, which are

based on popular OO patterns. This allows extension developers to implement their preferred architecture but is a drawback in consideration of maintainability, re-usability, and standardization. Joomla requires an MVC implementation within its main extension type (components) to adhere to the community's development conventions. Though, all APIs provide interfaces for event-based support. This allows developers to register actions which have to be performed during system events like user authentication or content preparation.

Table 2.5: API Support (2)

WCMS	Standard file structure	Decoupled extension configuration	Backend/Frontend support
WordPress	(✓)	✗	✗
Joomla	✓✓	✓	✓
Drupal	✓	✓✓	(✓)

The APIs of Joomla and Drupal request extension developers to implement a standardized file structure in order to ensure a homogeneous deployment to a running website. Joomla goes even further, requiring a strict separation of the file structure based on the API's MVC pattern. Additionally, Joomla provides conventions for all extension kinds. WordPress follows a more flexible approach since no conventions for a file structure exist. However, the WordPress community recommends a file structure which is popularly used.

Drupal's configuration approach, which was introduced with the latest major version, allows a sophisticated decoupled extension configuration. This brings some advantages since the actual code may stay unaffected during extension evolution. For instance, if the menu logic for the extension changes, developers must not change the code. Instead, they just have to change or augment the parameters in the configuration file. Joomla, also provides decoupled configuration files in order to follow a convention over configuration approach, but not in the sophisticated way as Drupal does. WordPress configurations must be placed within the actual PHP files in form of annotations, e.g. as part of DocBlocks.

Joomla allows to implement additional features separately for the desired section of the system (frontend/backend), which ensures a standardized and reusable extension structure. WordPress and Drupal also enable the development of features for a respective section, but delegate the responsibility for the structure to extension developers. So, a convention based on the extension mechanism can not be provided.

For an overall comparison, we sum up the score of the supported extensibility features based on the introduced criteria for community support, extension features, and API support. In order to quantify the score, we use the following relation: ✗ \rightarrow 0, (✓) \rightarrow 0.5, ✓ \rightarrow 1, ✓✓ \rightarrow 2. It must be mentioned, that the validity of the scoring should be confirmed in future work. This can be achieved, e.g. by interviews and surveys with industrial practitioners. To keep the comparison as objective as possible, the criteria is not weighted.

Table 2.6: Scoring of WCMS Extensibility

WCMS	WordPress	Joomla	Drupal
Developer Support	2	4	4
Extension Features	6	7	6
API	2	7	5.5
Score	10	18	15.5

Based on this score, Joomla stands out as the most suitable reference WCMS for an MDE approach. Especially the extension features, like the *separation into various extension kinds*, represents a well-suited reference implementation, since the extension mechanism of WordPress and Drupal are less complex. Additionally, Joomla comes with the strictest API conventions, based on coding standards and design patterns which have to be implemented by extension developers.

Another motivational aspect, which distinguishes Joomla from other WCMSs like WordPress, is based on its core implementation. Similar to Drupal, Joomla also uses extensions for the implementation of core features. So, even the core development may benefit from an MDE approach.

With this motivation, the idea of transferring the MDE approach of this work from Joomla to one or all of the other WCMSs is within the realm of possibility. Especially the presented and most popular WCMSs WordPress and Drupal should be suitable systems for further generalisation of the approach, since their extension mechanisms are similar in most cases or less complicated in comparison to Joomla. Requirement for the porting process is a matured modelling language, which generalizes the presented extension features regardless to the target WCMS.

2.3 Common Extension Development Scenarios

As previously described, extension developers face various challenges during extension development. Based on our experience in WCMS extension development, we identify three frequently occurring development scenarios:

- Development of independent extension like WordPress plugins or Joomla components.
- Development of dependent extensions like Joomla modules.
- Migration of existing extensions between different (versions of) CMS platforms.

These scenarios are confirmed by professional extension developers during conducted semi-structured expert interviews (see Section 4.1). In the following, we will describe these scenarios more detailed.

2.3.1 Scenario 1: Development of Independent Extensions

The first scenario, as illustrated in Figure 2.7, addresses the development of independent extensions to be used in a running WCMS instance (cf. [188]). Such extensions are particularly desirable during their evolution: If a developer changes the extension, no side effects due to dependencies occur in the system where the extension is deployed to. Examples for independent extensions are WordPress plugins, Joomla components, or Drupal modules which have no dependencies to other extensions. During evolution of other installed extensions, the extension should not show any side effects. The only dependency of such extensions is the use of the core API.

However, to ensure a correct interplay between the extension and the core of the underlying WCMS instance it is fundamental to comply with the development guidelines and make use of the respective core API. This requires additional development and testing effort, since even subtle errors like wrong file or class name patterns can lead to unexpected crashes that are not discovered until runtime (cf. problem statement 1). To tackle this challenge, extension developers typically copy existing extensions and adapt them to their requirements (clone-and-own practice). This, however, typically leads to oversized extensions including useless code. This challenge can be addressed by a model-driven approach providing tool support, e.g. by code generators which generate high quality extension code, which adheres to the code guidelines and implements the provided WCMS API.

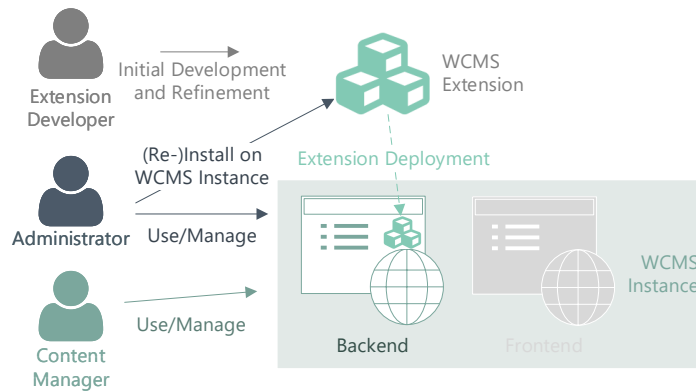


Figure 2.7: Development of Independent Extensions

The scenario occurs in two variants: First, the initial development of an extension and second its iterative improvement. Our scenario definition addresses both, whereas the initial development can be interpreted as the first iteration. Existing boilerplate generators can support initial development. Though, no tool support for iterative improvements exists (cf. problem statement 5).

2.3.2 Scenario 2: Development of Dependent Extensions

With the second scenario we introduce the development of extensions that depend on other extensions, by using some of their artefacts – a common practice to prevent multiple implementations of the same functionality (cf. [188, 186]). Additionally, this allows developers to augment existing extensions (e.g. third-party extensions) without changing their code base. We illustrate this scenario in Figure 2.8.

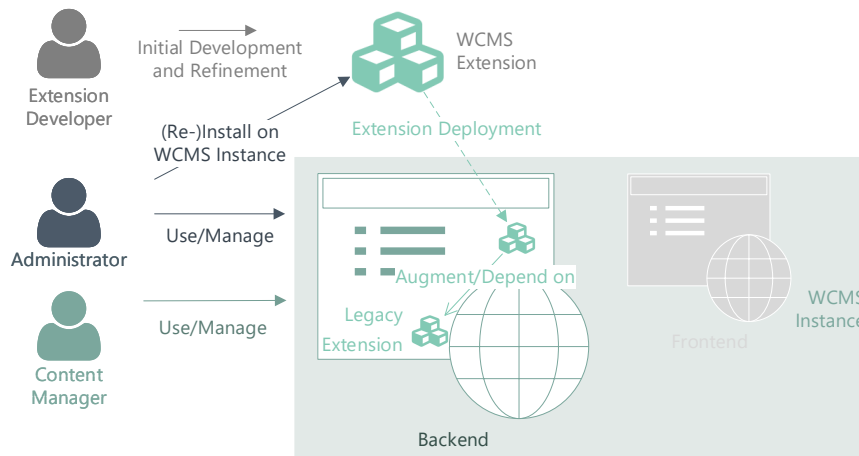


Figure 2.8: Development of Dependent Extensions

In WordPress, plugins may refer to other plugins, whereas Drupal allows interdependencies between modules. A common interdependency of Joomla extensions typically exists between components and modules. It is common practice to use artefacts of existing extensions within a component or module to increase the functionality of a Joomla system without developing software fragments anew. Components may reuse data access objects (DAO) or view templates from other components, whereas modules often use the database of existing components, since they usually provide no own data management.

In Figure 2.9 a common dependency between modules and components is illustrated. As previously described, a Joomla module uses the data of a Joomla component. To this end, two popular implementations of the dependency are presented. The first variant is based on a helper file in the module which features the use of a component DAO via reference, using inclusion methods of Joomla's singleton implementation (Figure 2.9a), whereas the second implementation directly accesses the components database by using SQL statements (Figure 2.9b).

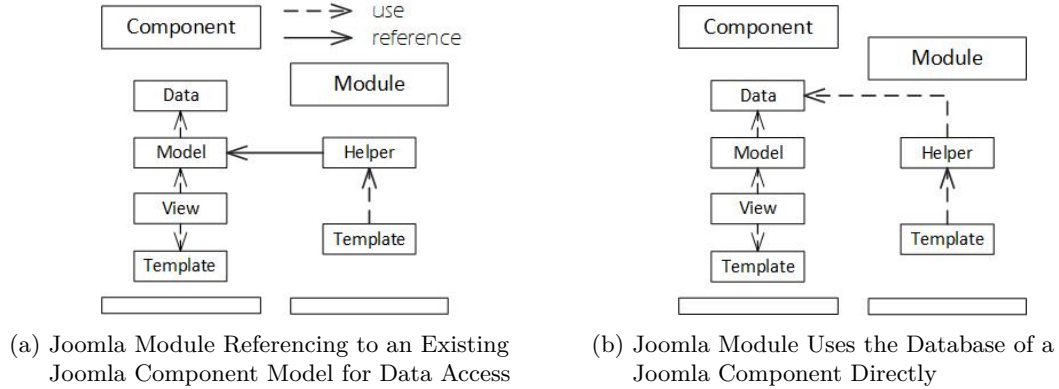


Figure 2.9: Common Dependencies Between Joomla Extensions (cf. [186])

During this scenario, developers are challenged by the required reverse engineering process which necessitates understanding implementation details of the base extensions (cf. problem statement 3). Moreover, evolving base extensions which are installed on the same WCMS instance present a challenge for administrators and developers. With every dependency the number of potential vulnerabilities increases (cf. problem statement 4). This challenge is not covered by existing tool support (cf. problem statement 5) but can be addressed by MDE, e.g. through specifying dependencies within extension models. Tool support for reverse engineering of the base extension supports developers to keep models of the dependencies updated.

2.3.3 Scenario 3: Migration of a Legacy Extension to a new Platform (Version)

In the third scenario, as illustrated in Figure 2.10, we address the migration of an existing (legacy) extension to a new WCMS platform or a new platform version of the same WCMS (cf. [187]).

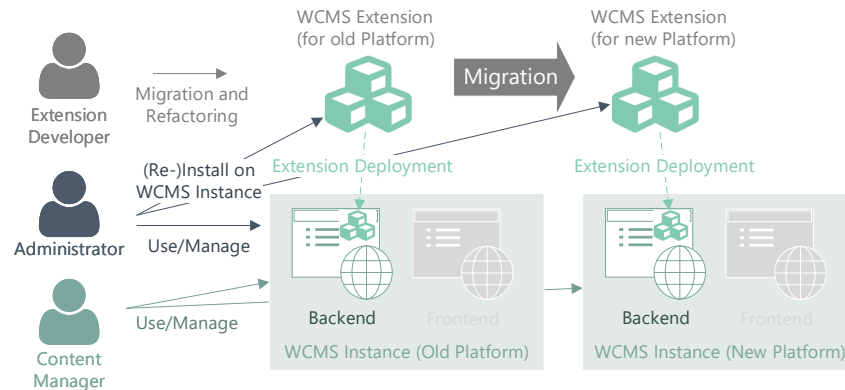


Figure 2.10: Migration of an Extension to a new Platform (Version)

Migrations to new platform versions of the same WCMS may require the same effort as migrations to another system. System enhancements, feature addition, and architectural reconstruction in order to increase the elegance, efficiency, and generalizability of a WCMS API is a worthwhile motivation for WCMS communities to release new major versions (cf. [17, p.21]). However, major version changes are typically characterized by tremendous changes of the core platform and its API, which usually break existing extensions. So, every new platform version forces extension developers to migrate their legacy extensions to the new API, to ensure their operability within updated WCMS instances. The required effort grows immeasurable, if the number of extensions to migrate rises (cf. problem statement 2). As experience has shown, missing documentation, non-existing tool support (cf. problem statement 5), and required effort often led to dying extensions. Often developers are not able to migrate their software in a reasonable amount of time. In this case, administrators have to replace their extensions which, in turn, is associated with additional effort. If no alternative extension exists, administrators are often obliged to keep their WCMS instance running with older platform versions until the required extensions are operable on it. By providing tool support for the migration of WCMS extensions to a new platform (version), the required effort could be reduced. An MDE infrastructure comprising reverse and forward engineering facilities allows support during the model extraction from existing extensions and the generation of extension code for the new platform.

Figure 2.11 illustrates an example for the differing file structures of two consecutive Joomla versions (3 and 4⁸). In Joomla 3, a view folder contains both the view class and all corresponding view templates whereas Joomla 4 requires a structure where both are completely separated artefacts. Such differences must be considered by extension developers in order to update their extensions to the new API specifications of the core system. Obviously, the same applies to the code level.

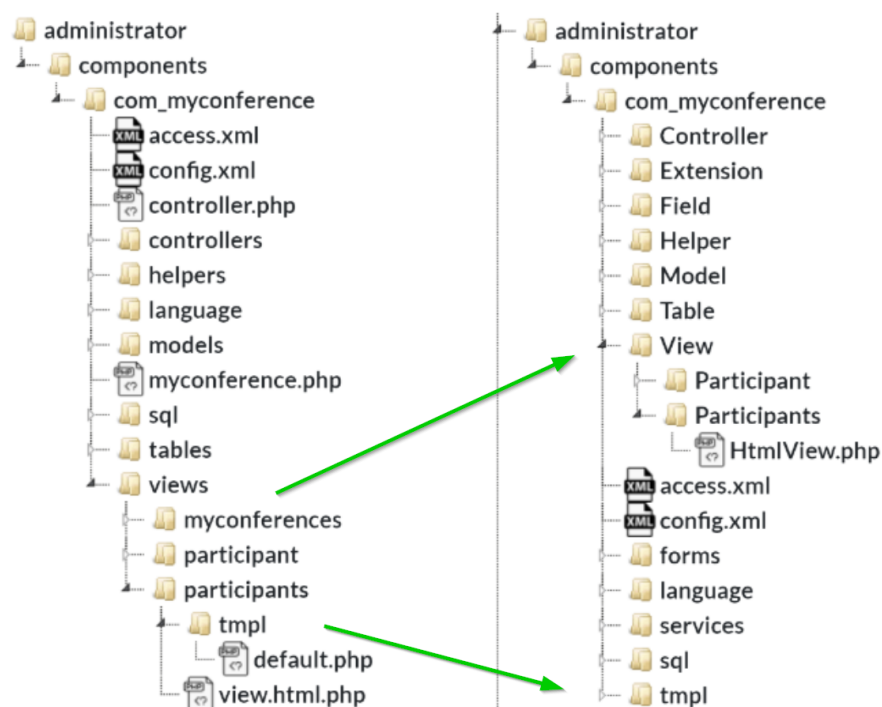


Figure 2.11: Differing file structure of a Joomla 3 (left) and a Joomla 4 (right) component (administrator folder)

⁸As of 2020, Joomla version 4 is successively re-developed due to its unstable state. The comparison is based on the Joomla version 4 Alpha 12 [167], which was the latest release while this thesis was written.

The migration scenario also includes the migration of extension data. Usually, data migration requires rather small effort, since the database structures mostly depends on the respective extension. After a platform update, this structure must not necessarily be changed for the new version. Often, a copy of the extension's data or a migration script is sufficient during the migration process. The APIs of popular WCMSs provide functions to run implemented migration scripts automatically during extension installation. This feature is typically used by extension developers for extension updates and can also be used for migration purposes. However, by following an MDE approach, these steps can be processed automatically by a code generator.

2.3.4 Further Scenarios

In addition to the previously described scenarios, two sub-scenarios should be mentioned due to their relevance (cf. Section 4.1). These scenarios mainly address already existing *legacy extensions which are not longer maintained* by their original extension developers but used within existing WCMS instances. Such extensions are often *partially augmented by new features* or *completely reengineered* by new extension developers. Additional possible scenarios or combinations with less relevance should be addressed in further research.

Scenario 4: Partial Augmentation of Legacy Extensions

A common practice in WCMS extension development is the augmentation of existing extensions with custom features (see Figure 2.12). This is a sub-scenario of scenario 2 (development of dependent extensions) since the new feature is an augmentation to an existing and typically deployed extension and usually depends on the original code or data. New features are added directly to the existing extensions by new views, new or refined database structures, or overrides (cf. Section 2.1). This is sufficient, if the extension is no longer maintained or the original developers refuse to add the desired feature.

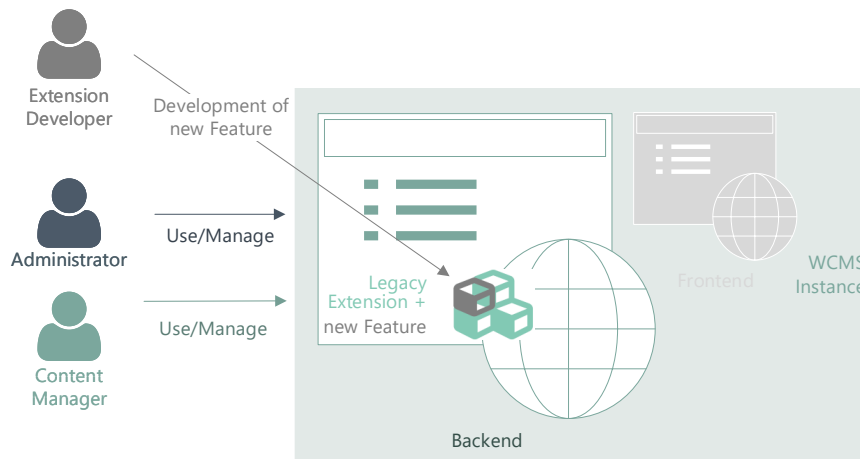


Figure 2.12: Partial Augmentation of a Legacy Extensions

To augment an existing extension, developers have to gain insight about the extension structure. Standardized extension structures facilitate such a discovery, but are not necessarily implemented. A challenge, developers have to face during the augmentation process is to check, if their refinements influence other dependent extensions (cf. problem statement 3). This is a fundamental requirement to avoid side-effects. Another challenge occurs, if the existing extension is augmented by custom code but contingent on further updates by the original developers which may override custom refinements (cf. problem statement 4). In this case, administrators

have to decide, if they refuse to update the extension or force developers to refine the updated extension anew. Both challenges can be addressed by an MDE infrastructure which comprises reverse engineering facilities for support during extension understanding and code generators which provide partial code generation for existing extensions.

Scenario 5: Modernization of Legacy Extensions

Another common extension development scenario addresses the code quality of legacy extensions. If an extension is not further maintained, but an essential part of a WCMS instance, developers often submit it to a reengineering process. This scenario can be declared as sub-scenario of scenario 3 (migration of a legacy extension to a new platform version), since the reengineering process is similar to the code migration process. However, the targeted goal differs. The migration scenario aims at porting an extension to a new platform, whereas the reengineering scenario is intended to modernize a legacy extension in order to refine it (scenario 4) or augment it by new extensions (scenario 2). The modernization scenario is illustrated in Figure 2.13.

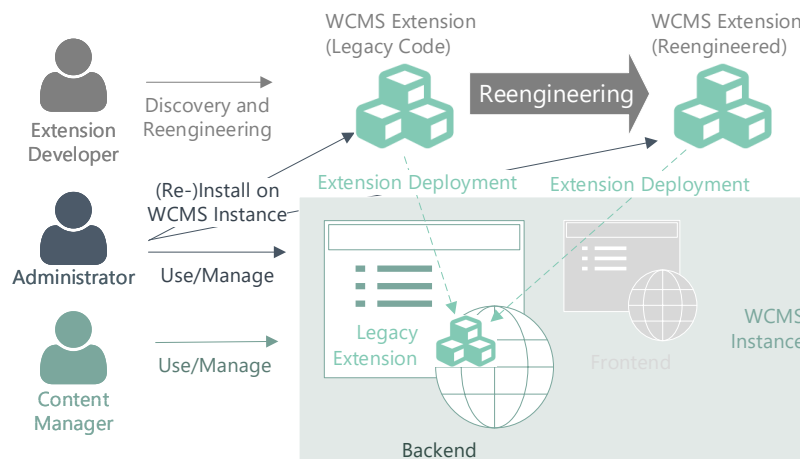


Figure 2.13: Modernization of a Legacy Extensions

Extension developers have to discover the structure of the legacy code in order to reengineer the extension to the new desired structure (cf. problem statement 3). Then, after the reengineering and re-implementation process, the resulting extension can be reinstalled to running WCMS instances or provided as new version, e.g. in an extension directory. The main challenge is to ensure that all features are provided by the new extension and the quality is not impaired. Additionally, dependencies to the legacy extension like depending extensions must be considered during the reengineering process in order to avoid errors during runtime. Similar to the previously described scenario, an MDE infrastructure can be supportive for extension developers during the reverse and forward engineering process.

3

Model-Driven Engineering

*The entire history of software engineering
is that of the rise in levels of abstraction.*

– Grady Booch

Being a software developer these days changed tremendously compared to the past. New emerging programming languages and frameworks necessitate a high potential of flexibility and willingness to learn. A promising approach to support developers dealing with complex and fluctuating technologies is to change the abstraction level of programming languages and frameworks. This allows the application of *model-driven engineering (MDE)* techniques to increase productivity.

During the last decades the use of models as abstract software representation became a common practice in software development. Usually, models are used to describe a software system in form of sketches (partial view) or blueprints (detailed specification) [28]. However, during the last years they established as representation of an executable program [28]. So, besides improving the understanding of a system and simplification of the communication between developers and customers, they can serve as main drivers of a software system. Some developers go even further and describe their programs as models what shows their relevance in today's development. Using abstract representations of software as input for code generators, which take over the work from development teams, seemed to be the next logical step in software development. Therefore, model-driven engineering practices enjoyed rising popularity in the last decade. Generating most of the code facilitates a standardized code base with better quality requiring less development effort in contrast to conventional software development.

Engineering software in a model-driven way usually requires, beside models, support by a set of tools, further called *MDE infrastructure*. These tools enable rapid software development with a minimum of required hand coding allowing a faster deployment of software artefacts [199]. An MDE infrastructure typically comprises tools for forward engineering processes such as model editors and code generators. Additionally, reverse engineering facilities, e.g. for model extraction from existing software can be included. By using frameworks such as the Eclipse Modeling Framework (EMF) complete model-driven engineering solutions can be implemented. To integrate such an infrastructure into the MDE process it is inevitable to support different development platforms such as common development environments (IDEs) or the platform-independent development using web technologies, such as web-based model editors. Though, the required development effort for necessary infrastructure tools often exceeds the required effort of conventional development methods. Especially, if the software system which has to be developed is too complex, individual, or is not intended to be changed in the future, the initial development effort for MDE tools can increase tremendously. This led to a collapsing popularity of model-driven approaches in the last years.

For a proper explanation of MDE, we open up the scope a little more. To this end, this chapter will give an overview of the most common terminology including a classification of MDE in contrast to the related terms MDA, MDD, and MBE. So, we can differentiate between them and clarify their relation to each other. Moreover, we introduce the fundamental artefacts of MDE infrastructures, present popular development frameworks, and discuss suitable processes for MDE tool development.

3.1 Terminology

Investigating literature and the web, there is a common misunderstanding or missing knowledge of the differentiation within the model-driven engineering context. Most authors talk about model-driven engineering, but in fact mean model-driven development (MDD) or even model-based engineering (MBE). Sometimes authors write about different abstraction levels in model-driven engineering without the knowledge of the specified standard called model-driven architecture (MDA). This sub-section will clarify these terms and presents an overview about their limitations and relationships among each other. According to [28], there is a containment hierarchy among MDA, MDD, MDE, and MBE which will be presented more detailed within the next paragraphs.

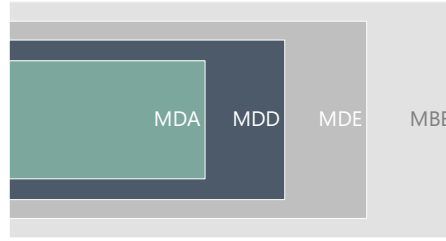


Figure 3.1: Relationship between MDA, MDD, MDE, and MBE (cf. [28])

MDA

Applying model-driven approaches during software development gained high popularity within the past decades. Therefore, the Object Management Group¹ (OMG) defined a formal specification in form of a meta-model which can be used as reference during model-driven software development - called *Model-Driven Architecture (MDA)*. MDA defines a flexible solution for software models (by means of UML² in the first instance). Instead of using large and complex models as monolithic artefacts, MDA recommends to split and arrange them within different levels of specification abstraction (see Figure 3.2).

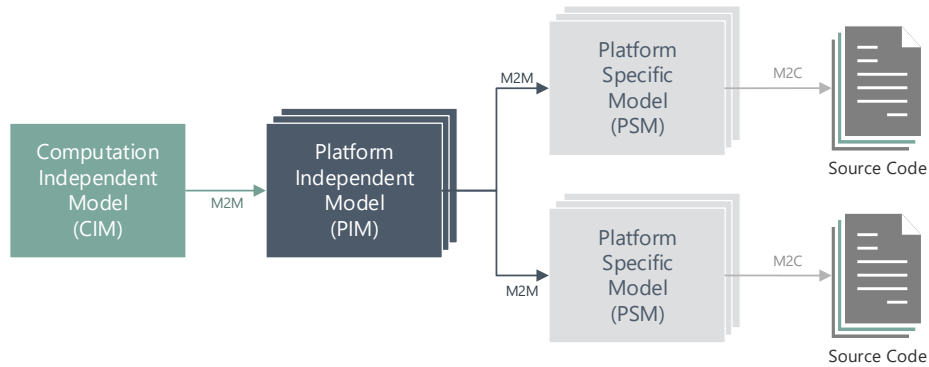


Figure 3.2: Abstraction Level Specification of MDA

The highest degree of abstraction in MDA is represented by the *Computation Independent Model (CIM)*. It is characterized by the colloquial description of system requirements. With the high degree of abstraction, the business processes can be defined regardless of the underlying software

¹The Object Management Group was founded in 1989 as a consortium to recommend standard specifications for object oriented systems. For further reading see [162].

²Unified Modeling Language, a general purpose language for the visualisation of software design on the basis of visual models.

systems, e.g. as a use case diagram. Usually, a CIM is used as abstract reference for a *Platform Independent Model (PIM)*. By a (mostly manual) model-to-model (M2M) transformation one or more PIMs can be created by a CIM. A PIM describes the functional aspects of a domain, independently of the underlying platform. This ensures a high reusability of the models. A PIM can be used as input for a (semi-) automated model-to-model transformation to create one or more groups of *Platform Specific Models (PSMs)*. These models describe the application with regard to the specific platform in an abstract way, but relatively close to the source code for the targeted platform. Like PIMs, PSMs are usually described with visual modelling languages such as UML or textual models such as XML.

The most specific artefact of the MDA standard is represented by source code. By means of a model-to-code (M2C) transformation, source code is generated, appropriate to a PSM or PIM. This transformation is usually executed by one or more code generators which contain the generic domain-specific information. For the case of individual code fragments, which can not be generated, protected regions or code models should be provided, to allow a subsequent augmentation of the code (see Section 3.3.5). However, this requires the consideration of adequate reverse engineering techniques in case of a further generation of the same software artefact.

MDA is generally adopted to different problem domains in form of several (UML-based) modelling languages and diagram types [28, p. 45]. Both perspectives are covered by MDA solutions.

MDD

Model-Driven Development (MDD) describes the development process, which can follow MDA, but is more motivated in the generation of software, whereby the models are the main artefact during the development. MDD is not a synonym for MDA, since MDA is a formal specialisation of MDD. MDD can be seen as a technical implementation of MDA using an MDD infrastructure. Typically, the term MDD is used for model-driven forward engineering of software including suitable modelling languages and tools for M2M and M2C transformations on the basis of the MDA specification.

MDE

The term which is mostly used within the model-driven context, as well as in this work, is *Model-Driven Engineering (MDE)*. MDE is actually a vague kind of MDD, since it includes the similar motivation for software development using models. However, in contrast to MDD, MDE also uses models as the main artefact for software evolution, documentation, testing, or reverse engineering. This means, MDE approaches may include MDD, but can go beyond the forward engineering process of model-driven software development. In related literature, such as [28], MD*E is often used as synonym for MDE, whereby the * is a placeholder for the actual field of application. For instance, MDSE (Model-Driven Software Engineering) describes software development by means of a model-driven approach. Another example is MDRE for Model-Driven Reverse Engineering. This allows a more concrete inference to the application of an MDE approach, even though the term MDE is often referred to MDSE. In this work, we use the term MDE as synonym to MD*E, since we address scenarios which include both forward and reverse engineering processes.

MBE

Within the context of model-driven software engineering, *Model-Based Engineering (MBE)* is often mentioned or even used as synonym for MDE. However, according to [28], MBE differentiates in its main motivation, since it does not describe a process for software generation based on models as main drivers. The term fits better to processes where models are used as supportive artefacts such as, e.g., domain models which help developers to understand a domain in order to create suitable applications with their conventional development method.

3.2 MDE in Software Development

Obviously, supporters of MDE usually emphasize its advantages but rarely mention its disadvantages and threats. Especially at the beginning of the MDE hype, only the advantages were considered in MDE-based projects. However, both sides must be investigated to get a clear picture of the topic.

Besides a *simplified and faster software development* the *flexibility of abstract models* is one of the main advantages. Models can be changed at any time and can be reused for several transformations. This allows both the *development for different platforms and the migration between them by using the same model*. Moreover, the creation of models enables a *better decoupling between domain and technology knowledge*. This, in turn, allows a better distribution of the software development tasks, tailored to specific roles like domain experts and developers with technology knowledge. Moreover, using abstract models for the formal description of a software system may increase its general comprehensibility, even for non-developers. So, a better communication between software developers and their customers can be enabled.

However, the application of model-driven engineering includes some downsides. A significant drawback of MDE approaches is based on the *high initial effort, required for the development of modelling languages, model editors, and code generators*. This effort is mostly underestimated during an MDE adoption and led to a downward trend of MDE at the late 2000s. As mentioned in [28], the drawbacks in industry which caused by failed MDE-based projects led to a chasm between software developers and MDE. One of the biggest difficulties is to convince developers to overcome their resistance to change and change their usual development habits.

Though, if advantages and disadvantages are clear, stakeholders of a software project can already decide in the preliminary stage, if MDE is a suitable solution or not. A glance at successfully adopted MDE approaches in industry (e.g. [16] and [94]) and case studies of MDE adoptions (e.g. [258] and [32]) indicates that MDE is situated on the *slope of enlightenment* stage in the Gartner Life Cycle³ [28, p. 19]. This means that an MDE approach, applied to an appropriate domain in a suitable project, can be a helpful and profitable alternative to conventional development approaches. In [233] the authors present the results of a survey with IT professionals considering the expected and fulfilled benefits of adopting MDE approaches in industry. The result of this survey emphasizes aspects like improved documentation, design and maintenance support, and standardization as fulfilled benefits, whereas platform independence seems to be rarely relevant. Quality and productivity are at the borderline of the fulfilment rate of the responses, due to the amount of hype around MDE within the last years. The conclusion of the authors sum up, that, if practitioners focus on a smaller set of benefits, MDE approaches become more successful. It became clear, however, that the MDE adoption must be intentionally of pragmatic nature. This means that not every detail has to be modelled to successfully adopt a model-driven approach. The appropriateness of a problem domain for a model-driven engineering approach must be further researched, though. This thesis will incidentally contribute to this research in the domain of WCMSs and the development of WCMS extension.

3.3 MDE Infrastructure Development

Adopting a model-driven approach, as part of a software development process, requires a set of tools to support developers. These tools must allow the creation and validation of models as well as their transformation to instances of other modelling languages or to code. Additionally, the tools should support developers in terms of versioning, testing, and integration of generated artefacts into the whole software system. Especially, if model-driven engineering constitutes a

³See [76] for a further explanation of the Gartner Hype Cycle.

subsystem, the latter determines its success within the whole project. Speaking of the required set of tools, the term MDE or MDD infrastructure has been established over the last decade, even though the term low-code platform has become more commonly used during the last years. The latter makes more sense for platforms which use textual models as the main development artefacts.

Developing such an MDE infrastructure requires a suitable development process to consider all the described artefacts. Due to the fact, that these artefacts must comply with the requirements of the respective problem domain impedes the development process additionally. To address the development challenges of MDE infrastructures, some best practices have been emerged in the last decade. The guidelines as presented in [221], [28], and [132] are typically followed, even though they do not specify standard procedures for the systematic infrastructure development. In [244] the authors present an agile bottom-up development process which we apply within this work. This process incorporates popular best practices into iterative development, what is typically the most suitable practice for MDE infrastructure development.

Within the next subsections we will give an overview of MDE infrastructure development, considering the design of domain-specific modelling languages, transformation techniques, meta-tools, IDE and custom code integration, and suitable processes for MDE tool development.

3.3.1 Design of Domain-Specific Languages

A fundamental decision at the beginning of introducing an MDE approach, is the choice of a suitable modelling language to be used within the development process. Since models are the main artefacts of an MDE approach, the *language must fit to the problem domain to the same extend as to support model designers during model creation* in the most appropriate way. Sometimes, a new DSL for a problem has to be developed. Since DSL development is a hard and time-consuming process, an extensive decision phase should be placed in advance (cf. [135]).

The main task during specification of a DSL is the *identification of domain concepts* and their *integration as domain-specific language elements*. The abstract definition of these elements, including their relationships and dependencies, represent the abstract syntax of a DSL [135]. By applying meta-modelling techniques as presented in [154], [135], [8], and [28], *the abstract syntax can be extracted from existing domain knowledge*. This includes existing legacy software implementations and documentation (reference applications) as well as knowledge of domain experts (developers and users). To gather knowledge from domain experts, empirical studies like interviews and surveys can be conducted. This analysis phase should result in a domain model which contains the terminology of the domain with relationships and dependencies of the domain features (cf. [154]).

In accordance to the DSL construction patterns as defined in [154], the *analysis* step is typically followed by a *design* and *implementation* phase - provided that the analysis is not leading to reconsider the decision of implementing a new DSL. DSL developers have to determine the formal nature of the language, i.e. if the DSL will be available as stand-alone language (*external DSL*) or shall extend an existing (modelling) language (*internal DSL*). A popular realisation of the latter is represented by UML profiles which extend the UML specification. Based on the design decisions, the language can be described formal or informal in order to obtain a meta-model for the domain. In [35], the author states that a meta-model is the main artefact of a DSL, "[...] somehow equivalent to defining the grammar of a textual DSL". Following a meta-model-centric design (cf. [28]), allows the use of a meta-model as base for further DSL implementations like language constraints, concrete syntax, and model transformations. *Language constraints can be used to limit the possible models* which can be created with the modelling language. So, more complex validation rules can be specified. The most popular language for constraint definition is the Object Constraint Language (OCL) [254].

Since the (in)formal definition of a DSL is useless for actual modellers, the implementation of a concrete syntax is inevitable. The *concrete syntax defines the actual notation of the defined concepts of the abstract syntax* within model editors. Whether the models shall be represented in a more visual or textual form, depends on the respective needs of the problem domain and tasks to be executed. While some developers prefer *visual models (modelware)*, which usually give a better overview of a (sub-) system, other developers would always choose a *textual representation (grammarware)*, due to its compact and structured form. Modelware approaches explicitly require concrete syntax definitions, whereas in grammar-based DSLs the abstract and concrete syntax can be defined in one step during grammar definition. A popular framework for the development of grammarware is represented by Xtext [60]. Based on the concrete syntax, which can be specified within an Xtext grammar a text-based editor can be generated automatically. Usually, such an editor must be explicitly implemented for modelware approaches by infrastructure developers.

3.3.2 Providing Transformations

As proposed by the MDA standard, it is useful to work with models of various abstraction degrees. By applying transformation on these models, model information can be transferred to models of other languages or more concrete models in both the same and other languages. MDA specifies the possible transformations as described in sub-section 3.1 (cf. $\text{CIM} \rightarrow \text{PIM} \rightarrow \text{PSM} \rightarrow \text{Code}$). Transformations can be applied as *model-to-model (M2M)*, *model-to-code (M2C)*, and *code-to-model (C2M)* transformations. A C2M transformation is typically used for reverse engineering purposes for existing software, whereas M2C is the common transformation in a forward engineering process. M2M transformations are mainly used for semantic mappings between different abstraction degrees of a system (vertical mapping) or between two systems (horizontal mapping). By applying these transformations the interoperability⁴ between concerned software artefacts of different formats can be achieved.

Model Discovery

In order to discover model information from existing code, e.g. for modernization of legacy software, a transformation must be performed either manually or (semi-) automatically. The latter requires the implementation of model discoverers which must be able to parse the code-format and generate a model which conforms to the meta-model of choice. Typically, the result of a model discovery process is a platform-specific model, which enables a further model transformation. Though, the model should include all aspects of the given code in order to fulfil the completeness of the discovered software. However, this requires sophisticated discoverers for the domain which have to incorporate all possible language features of the given code.

In contrast to conventional parser development, various frameworks for the implementation of model discoverers became popular during the last decade. Most of them implement the Abstract Syntax Tree Metamodel (ASTM) by OMG [161]. This meta-model describes a standard for the re-use of abstract syntax trees independent of the input language. So, the interoperability between various systems can be increased as well. One of the most popular frameworks for model discovery of Java-based projects, which follows the ASTM standard, is MoDisco for Eclipse [58, 29]. Even though the framework comes with a set of discoverers mainly for Java, MoDisco can be used for any legacy system due to its extensibility by own discoverers. However, creating such a discoverer requires the same amount of maintenance effort as conventional parser development. Hence, the decision of using a framework like MoDisco or implementing an own model discoverer depends on the problem domain and the related technologies.

⁴In the MDE context, the term *model-driven interoperability (MDI)* has become established as concept based on injections (C2M transformations) and extractions (M2C transformations). For further reading see [28, p.34ff].

Model Transformations

To allow interoperability between heterogeneous systems, model transformations can be performed. So, the semantic information of a model can be transferred to a model which conforms to another meta-model. This allows development scenarios like migrations from one system to another by using models as the main artefact. In addition, model transformations are typically used for abstraction purposes. On the one hand, model transformations can be used for the specialization of a given model, e.g. to augment models with platform-specific features. On the other hand, model transformations are often used for the generalization of models e.g. during model understanding in a reverse engineering process. So, resulting models of a model discovery process can be transformed to platform-independent models describing the problem domain independently of the underlying platform.

Model transformations can be executed within the same modelling language (*endogenous*) or between different modelling languages (*exogenous*). Typically, the result of a model transformation is a new model of the target modelling language (*out-place*). However, especially for endogenous transformations, model-refinements on the input model have been established over the last decade. Such *in-place* transformations are mainly used for model refactoring purposes. In [153], a more detailed taxonomy concerning model transformations is given. The Atlas Transformation Language (ATL) [117, 116] has reached high popularity for exogenous and endogenous out-place transformations, whereas graph transformation languages like Henshin [12] can be used for endogenous in-place transformations.

Code Generation

A code generator typically creates concrete software implementations using a set of programming languages for a specific domain. To this end, generators typically use input models which contain abstract information about the artefacts which will be generated. This brings a variety of advantages over conventional software development. Standardized and understandable code, replaceable underlying technologies, reuse of legacy code, and the possibility of partial code generation are just a few to mention. Brambilla et al. present a more detailed list of code generation benefits in [28, p29 ff.].

Designing a code generation approach requires the choice of an appropriate generation technique. Stahl et al. present common generator techniques in [221, p. 186 ff]. Summarized, they compare common generation techniques which differ in time of compilation, use of (meta-)models, and interaction of existing and generated code. Most of these generation techniques became popular and are naturally applied in present software development: *Generating documentation and tests based on code annotations like docblocks (code attributes)*, *filter-based template generation using XML and XSLT*, and *aspect-oriented code generation*, e.g. injection of loggers and tracers into class methods (code weaving). Based on these techniques, literature proposes alternative approaches for the creation of code generators, whereas the most common one in the MDE context is the realisation of *template-based* approaches which are construed for the interplay with instances of meta-models (cf. Figure 3.3).



Figure 3.3: Template-based Code Generation (cf. [221, p. 189])

Code templates typically apply to the structure of a meta-model instance in order to use the model information for code generation. This is done by mixing static code fragments with gaps for schematically-recurring code information which comes from the input models (e.g. class identifiers). The latter describes the part of template which depends directly on and is filled with code information during code generation. In order to create human-readable code templates, which are close to common code conventions in the problem domain, existing implementations are typically used as reference during template definition. A popular solution for the realisation of template-based generators with meta-model support is provided by the Xtend framework [59] which perfectly fits to grammar-based DSL solutions created with Xtext [60]. Both will be explained more detailed in Section 3.3.3, since they are relevant for the MDE infrastructure implementation part of this work.

Template-based generators are often implemented as monolithic software with no need for variety. However, in order to handle variability in code templates, a common procedure is to reuse existing templates by copy&paste and adapting the new templates [114]. A more elegant way is described in [81]. The authors present an approach for enabling variability in code templates by using so-called *variability regions*. So, code templates can be used within software product line engineering⁵.

Usually, code generators are used as part of MDE approaches, where model information is required during generation time. The resulting code can be deployed to the target system and can be replaced by re-generated code after model refinements. If a more flexible use of models is required, e.g. if the deployment of the generated code cannot be done easily or partial changes should be allowed, *model interpreters* have been proved as an alternative. Model interpreters allow the use and refinement of models during runtime by model parsing functionality within the actual code. In the domain of mobile applications, model interpreters are promising artefacts within MDE approaches. Typically, generated code becomes compiled to the native platform and is then be installed on the mobile device. Every change of the model requires a new generation, compilation, and deploy step for the whole application. A partial replacement of a generated fragment is not possible. By using model interpreters, model changes can partially influence the application without the need of a re-installation. However, both approaches can be combined as hybrid approach, e.g. by using platform-independent models for code generation and platform-specific runtime models. So, the heterogeneity of the various target platforms in the mobile application domain can be flexibly addressed during installation as part of individual runtime models whereas the actual application code is generated based on platform-independent domain models. This is another advantage of model interpreters. In [243], the author presents a successful realisation of a hybrid approach as part of an MDE infrastructure for the development of mobile applications with context support. The presented infrastructure consists of a template-based code generator for applications which can interpret runtime models once deployed to a mobile device.

In the WCMS domain, model interpreters are not mandatory since the common technologies allow partial replacement of files (e.g. after a re-generation). Additionally, the systems are rather homogeneously in comparison to other domains. The software to be developed is mainly based on interpreter languages. It seems, that the suitability of code generators in contrast to model interpreters correlates to the target domain inversely. Code generator approaches are more suitable for interpreted languages whereas model interpreter approaches fit in domains of compiled languages. However, a hybrid approach is always possible and the generator/interpreter design should be decided based on the required flexibility and needs of the targeted software and development process.

⁵Software product lines comprise sets of “software-intensive systems sharing a common, managed set of features that satisfy needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way” as Clements and Northrop state in [48]. Common artefacts in SPL engineering are *feature models* which consist the features and variants of a software system.

3.3.3 Support for MDE Tool Development

If developers decide to adopt MDE within a software project, lots of matured MDE tools are available nowadays. Especially Java developers are supported by a wide variety of tools, including model editors and sophisticated code generators. Even the web domain and embedded systems are domains which enjoyed much attention by providers of MDE tools. Besides commercial and proprietary solutions, free and open source tools exist to support MDE. The most popular representatives are Enterprise Architect [218], Matlab and Simulink [149], Rational Rhapsody [96], ObjectiF [155], and MetaEdit+ [131, 215]. For further reading, find a comparison of actual MDE tools in [123] and [122], whereas the authors of [120] compare and evaluate current tools.

The *Eclipse Modeling Framework (EMF)* [83] stands out by its variety of solutions for different MDE use cases and MDE-based problems. Since it is open source, developers built a wide set of tools for modelling, validation, transformations, model versioning, model merging, reverse engineering, and MDE tool development itself. Even though all of the tools are initially developed for the Eclipse IDE, other IDEs such as IntelliJ IDEA from JetBrains are also supported. Figure 3.4 illustrates the most popular EMF tools grouped by their intended area of application.

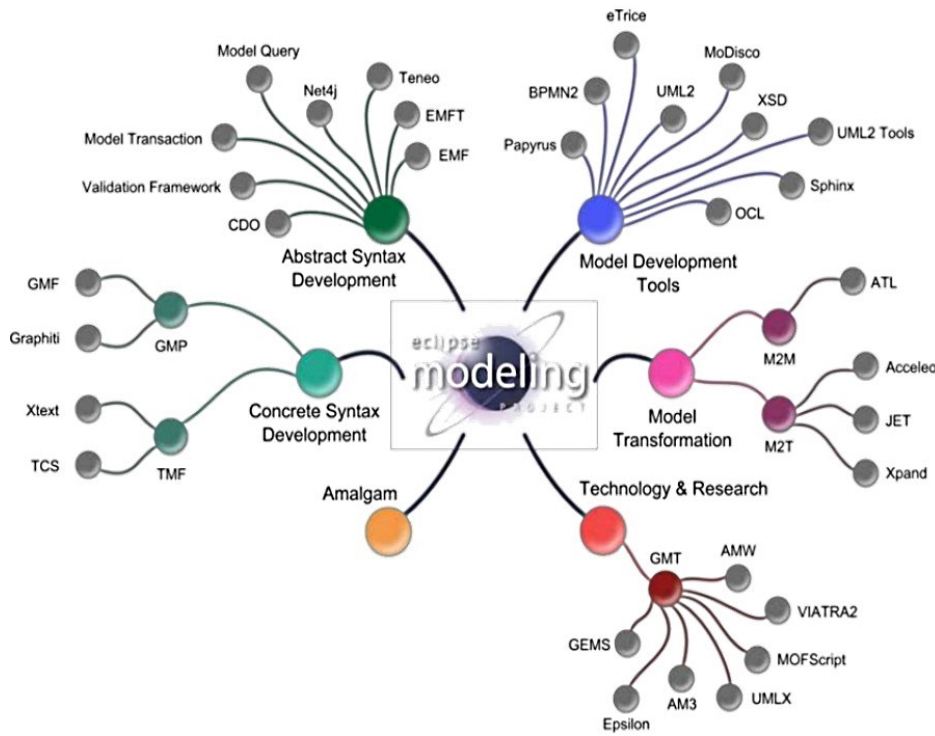


Figure 3.4: MDE Tools based on EMF [38]

EMF comes with a powerful meta-modelling language called *Ecore*. This language is an implementation of the (E)MOF⁶ standard, defined by the OMG. In addition to the Ecore language, EMF provides tool support for the generation of Java-based API code to enable access to instance model information within Java programs. In order to allow the creation of instance models for individual Ecore-based domain models, EMF provides a tree-based instance editor which contains various reflection and validation features.

⁶The Meta-Object Facility standard represents a four-layered scheme, which describes the relationship between objects (M0), their description as models on a more abstract layer (M1), and their specification by meta-models on the next abstraction layer (M2), which in turn are described by meta-meta-models (M3). A popular implementation of the M2 layer is the UML meta-model, which can be used to create domain models on the M1 layer. For further reading see the specification by OMG in [165].

One of the biggest advantages of EMF is its support for creating own formal specifications for models in form of meta-models and domain-specific languages (DSLs). A prominent EMF-based tool set is the combination of *Xtext* [60] for DSL development and *Xtend* [59] as sophisticated and integrated template engine for creating code generators. *Xtext* allows the creation of domain-specific modelling languages as grammars in an EBNF-like style. Based on an *Xtext*-based grammar, *Xtend* can be used for the straightforward implementation of code generators which use instances of self-described DSLs. The *Xtend* programming language allows the creation of code templates embedded into a Java-like syntax which is generated automatically to Java code during template implementation. Plain Java code is also allowed within *Xtend* templates, since the language is a superset of Java.

Based on a grammar definition with *Xtext*, a set of Eclipse plugins can be generated, comprising a text-based editor for the creation of model instances and a skeleton test infrastructure for automated tests of both the grammar and generator. In addition, a web-based application can be generated, which consists of a text editor for model instances. All generated editors come with syntax highlighting, auto completion, and model validation. To enable individual enhancements, the generated plugin structures allow the implementation of additional features like editor quick-fixes, validation rules, and formatters. The generated skeleton files allow a straightforward entry for infrastructure developers. So, DSL, editor, and code generator development can take place simultaneously. This allows a more rapid and agile development of MDE infrastructures. Regardless of the formal description of a DSL, EMF allows the (de-)serialization of operations to open formats such as the XML Metadata Interchange (XMI) format. This allows tool developers to create reverse engineering facilities like model extractors in combination with a self defined DSL in an elegant way. Therefore, the *Xtext/Xtend* combination has been firmly established during the last decade. In [24], the authors present how *Xtext* and *Xtend* can be used together to build powerful MDE infrastructures.

Even though EMF comes with some drawbacks⁷, based on the Eclipse IDE itself or missing documentation, the framework convinces by its matured state, extensibility, and versatile application. Therefore, EMF is the MDE infrastructure development platform of choice in this work. It assists tool developers to keep down the initial development effort and allows a straightforward development and maintenance of MDE infrastructures.

3.3.4 IDE Integration

A drawback of EMF and EMF-based frameworks is the limitation to the open source Eclipse IDE. Even though Eclipse enjoyed huge popularity in the first decade of the 2000s, it is replaced by more modern IDEs like IntelliJ IDEA, Microsoft Visual Studio Code, or platform- and language-specific IDE solutions like WebStorm, Android Studio, or XCode. However, due to its large community and the availability as open source project, Eclipse is still popular in academia and smaller companies. Another trend shows the popularity of web editors or so-called cloud IDEs, which can be used platform-independently.

To support as many developers as possible, various IDE solutions should be considered during the development of MDE tools. Otherwise, they cannot exploit the most possible MDE potential. Studies as the one presented in [258] by Whittle et al. have shown, that potential developers decide against a model-driven approach due to the tool integration into their common tool chain.

In this work, we consider both the development of IDE-based MDE tools such as Eclipse plugins for modelling and code generation as well as their integration within a platform-independent cloud IDE.

⁷See the work of Kahani et al. in [121] which presents results of a research addressing the barriers and common problems of EMF.

3.3.5 Custom Code Integration

The main idea behind MDE is to deal with models and generate application code without the need of handwritten code. However, based on empirically evidence, this is not possible in real-world scenarios. A significant challenge of MDE adoptions is the integration of custom code with generated code. Therefore, various approaches became popular in order to address this challenge.

A widespread integration approach is the addition of *protected regions* to generated code [221]. These regions, which are typically introduced by an explicit comment section in the code, are interpreted by code generators during a re-generation step in order to omit overwriting the file or area during re-generation. So, a durability of custom code can be ensured. Protected regions can be easily included into code templates and their use is a straightforward approach, if iterative development is not required, e.g. during initial scaffolding. However, the use of protected regions comes with several drawbacks during long-term MDE adoptions. If a generated file is augmented by handwritten code it is usually checked-in in a version control system like git. However, generated files should not be checked-in, since they are disposable artefacts in contrast to models [250]. Generated files which are kept due to including custom code may become outdated since they are not affected by model changes during iterative development. This, in turn, may lead to additional refactoring effort in order to achieve clean code.

In accordance to the recommendations of Stahl and Völter in [221] and [250], generated and handwritten code should always be kept in separated files or, better still, folders. In order to integrate generated and custom code, *extension points* should be used. This includes the implementation of suitable design patterns, such as the generation gap pattern [249, 71], as well as prominent concepts like abstract classes and interfaces, dependency injection, or the use of callbacks. The kind of extension point depends on the composition features of the target platform and programming language. In the scope of this work, extension points are the most suitable approach in order to compose generated with handwritten artefacts. In Section 6.2.2, we present a code generator for WCMS extensions which augment and can be augmented by handwritten extensions without mixing generated and handwritten code.

In addition to protected regions and separated files, custom code can be directly added on model level, e.g. as *code models*. By linking models with custom code, e.g. by model annotations, code generators can paste custom code directly into the generated code, without the need of customizations after the generation process. This is an advantage for iterative MDE processes including repetitive code generation. The lift of custom code to the model level is an alternative to protected regions, if custom code integration cannot be achieved by extension points, e.g. if the target platform does not support modular architectures.

3.3.6 Development Process

Developing an MDE infrastructure is equal to the development of any other software. Therefore, infrastructure developers have to make the same decisions as application developers. This includes the choice of an adequate development process as well as the tools to be used during this process. Modern and flexible development methods such as Scrum, Kanban, XP, and Feature-driven Development have become the standard development practices nowadays. All these methods follow the agile manifesto [21], which defines the most valuable aspects of software development in agile processes. In conclusion, the main principles of agile development methods are early delivery of working software, and quick response to changes within the customer collaboration. In the case of MDE infrastructure development, customers are developers who should use the implemented MDE tools. Following an agile development process, achieves rapid feedback and the provision of infrastructure components at early development stages, considering the feedback within further development iterations. Fortunately, a variety of supporting

frameworks exist, to allow rapid MDE infrastructure development. Meta frameworks like Xtext and Xtend can be utilised to create an MDE infrastructure applying a model-driven development approach during tool development. So, the advantages of *MDE and agile development can be combined to increase the success of the approach* (cf. [221, p. 80]). As experience has shown, this has been worked out well within actual projects, such as presented in [269] and [140].

In [73] and [84] the authors compare the development of MDE infrastructures with *domain engineering*⁸ (also called product line engineering). Figure 3.5 illustrates the domain engineering concept in the application engineering context.

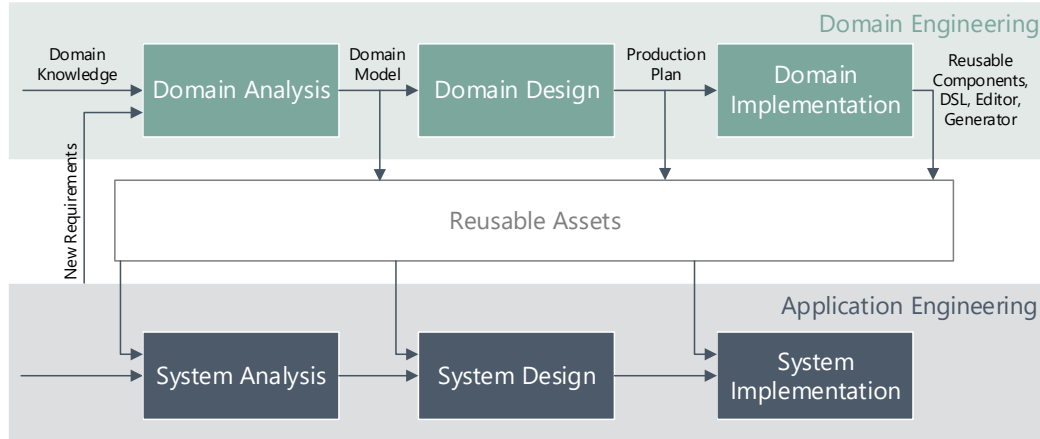


Figure 3.5: Domain Engineering and Application Engineering (adapted from [84])

Domain engineering is divided into three main process parts: *Domain Analysis*, *Domain Design* and *Domain Implementation* [84]. To ensure sophisticated MDE infrastructure assets, such as model editors and code generators, they have to encompass domain knowledge as much as possible. Ideally, fully developed frameworks with standardized functionality are available. To achieve this goal, infrastructure developers have to analyse a set of reference applications to extract the required domain knowledge. The common processes during the development of adequate reference applications are *System Analysis*, *System Design* and *System Implementation* [84]. In every process, the application developers can use the current version of the MDE infrastructure and give new requirements to the infrastructure developers.

This process, also called *bottom-up development* [15], represents the most proper way to develop an MDE infrastructure. However, the resulting infrastructure can usually only be used for a specific version of the application to be developed. In the case of rapid evolving software domains, the process has to be repeated over the whole life span of the infrastructure. Initiated by new user demands and changes in the underlying technology, components like DSLs or code generators have to be refined or further developed. These changes, in turn, may affect dependent artefacts such as model editors, showcase models, and reverse engineering facilities.

In [244], the authors propose an *agile bottom-up development process*, which addresses the challenges during infrastructure development for rapid evolving domains. This process is built on three steps: *domain analysis*, *continuous DSL and tool development*, and *adaption of related artefacts*. The steps will be described more detailed below, since this process is adopted during the development of the MDE infrastructure for WCMS extensions which is presented in Chapter 5 (DSL) and Chapter 6 (transformation tools).

⁸The process of systematic investigation of a domain with focus on high reusability of the gained domain knowledge [52].

Domain Analysis: At first, a domain analysis has to be performed in order to *develop initial versions of infrastructure components*. This includes the extraction of schematically repetitive, generic, and individual fragments from existing reference applications. The resulting domain concepts can then be integrated into a DSL and code generator templates. The latter also includes the generic code fragments from the reference applications. During the code generation step, the generator must merge the generic code with model information based on the used DSL. Figure 3.6 illustrates the domain analysis procedure.

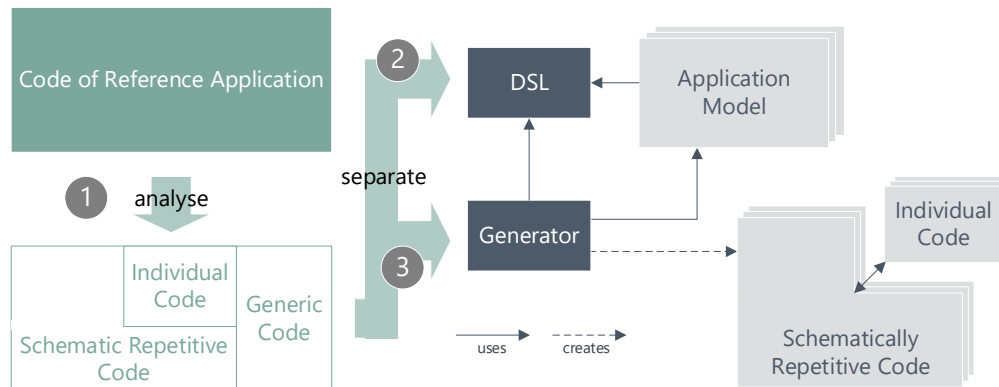


Figure 3.6: Domain Analysis Concept (adapted from [221])

The identification and investigation of adequate reference applications represents the initial step (1) of the domain analysis. During this step, it must be ensured that the reference code follows certain quality guidelines and implements suitable design patterns. Additionally, common anti-patterns and code smells have to be identified, to address them during the next steps of the domain analysis. This step can be supported by static code analysis tools. Examples for such tools in the WCMS extension domain are PHPCodeSniffer [220], PHPMD (PHP Mess Detector) [180], and JSLint [51].

Within the second step (2) of the domain analysis, the domain concepts are determined. These concepts can directly flow into the DSL definition by abstracting schematically repetitive parts. To identify repetitive code, clone detection tools can be used. To this end, promising approaches for mining exact and near-miss code clones [271, 115, 224] exist. Comparisons and evaluation of code clone or source code mining techniques have been conducted in [198] and [134].

In the third step (3) the code generator templates are specified. These templates comprise generic code fragments and gaps for specific model information. During code generation, the gaps are filled by the generator, using the application-specific information of the input model (DSL instance). Even though first approaches arise, no tools addressing the automatic extraction of generator templates based on reference applications, seem to exist.

Continuous DSL and Tool Development: The second step of the process addresses the *evolution of the MDE infrastructure*, triggered by new requirements or changes of the underlying technology. In accordance to the statements of Vaupel et al. in [244], infrastructure components like DSLs, model editors, and code generators should be developed in fine-grained iterations (cf. Figure 3.7).

If a new feature is determined to be supported by the MDE infrastructure, a prototype of the application has to be generated and manually extended by the new feature. To this end the application developer has to analyse the generated code and incorporate the new feature homogeneously with the generated code. This, in turn, requires high quality of the generated code. To ensure working application code with good quality, tests could be performed before and

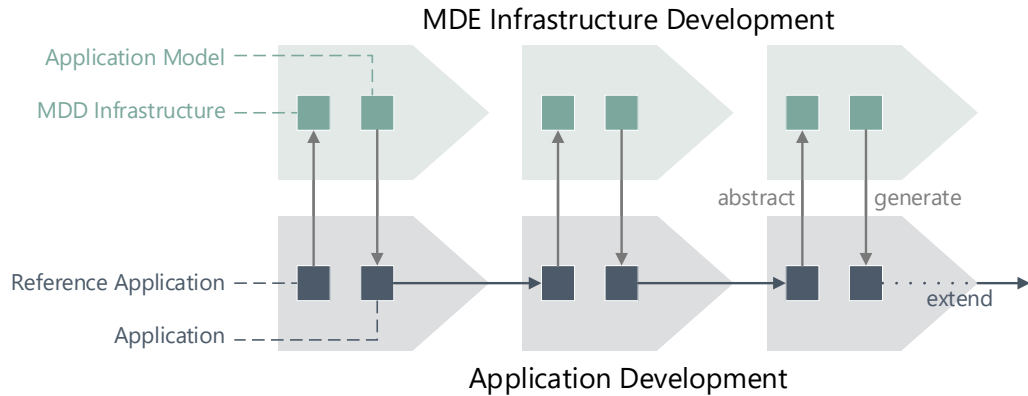


Figure 3.7: Agile MDE Infrastructure Development Process in Action
(adapted from [244])

after the addition of the new feature. So, the procedure can be adopted within agile processes like test-driven development. The infrastructure developers can then analyse the new code and augment the infrastructure artefacts by the new feature in a synchronous manner. The same applies to both DSL refinements and changes in respective generators. If tests were used during the manual implementation of the feature, the same tests could be used after developing the new feature with the refined infrastructure. Every iteration includes the domain engineering steps which are illustrated in Figure 3.5.

Besides the addition of the new feature to the DSL and generator templates, according MDE tools like model editors and reverse engineering facilities must be adapted as well. To maintain the MDE infrastructure, a variety of meta-tools exist. If tools like Xtext are used for the development of a textual DSL, the textual model editor can be automatically generated anew based on the DSL. However, if additional features like validators, formatters, or quick fixes are part of the editor, they have to be refined as well. If the model-driven process consists of model-transformations, tools like ATL [69] or Henshin [138] can be used for the definition and refinement of transformation rules.

Adaptation of Existing Application Models: The third step focusses on the *migration of instance models after a refinement of the DSL*. To keep the models consistent, all affecting changes in the DSL must be identified and updated in the existing application models. A restriction to allowed DSL changes which do not affect the existing instance models is not preferable, since it compromises a maturing evolution of the DSL. Using DSL-specific validators allows a manual refinement of existing instance models after DSL changes. However, to minimize the required effort, approaches presented in [75], [91], and [46], should be considered during the adaptation process. The mentioned approaches contain well-defined and reusable operators and migration steps which can be performed independently to the problem domain or target platform. Additionally, they present approaches for an automatic co-evolution of DSLs and instance models. To enhance the co-evolution, M2M techniques like [130] should also be considered during the infrastructure development as well. Other tools in this context like Eclipse Edapt [90] and Epsilon Flock [228][195] require mapping configurations for applying migration operations. These configurations can comprise predefined (reusable) operations or manual defined migration scripts. To handle model versions (e.g. for fall-back strategies), tools like EMFStore [88] can be used. Furthermore, tools like EMFCompare [227] and SiLift [129] allow model comparison, an essential feature to support version management.

4

MDE of WCMS Extensions - General Solution Concept and Requirements

*The most important single aspect of software development
is to be clear about what you are trying to build.*

– Bjarne Stroustrup

In order to research the profitability of model-driven development in the WCMS domain, we propose a concept for MDE of WCMS extensions, based on the common development scenarios in the domain. To this end, we refer to the presented extension scenarios from Section 2.3. This includes the investigation of how MDE can be integrated into the development process during these scenarios and helps to identify the necessary requirements to achieve this goal. In this chapter, we aim to answer *RQ1: How can MDE support common WCMS Extension Development Scenarios?* To address this question, we investigate three follow-up question:

RQ1.1: How relevant are development scenarios 1-5 for industrial practitioners?

RQ1.2: How can MDE support developers during these scenarios?

RQ1.3: Which facilities are required to achieve MDE during these scenarios?

To this end, we confirm the previously described challenges (cf. problem statements 1-5) and development scenarios of WCMS extension development by the results of conducted *semi-structured interviews with domain experts*. Additionally, we determine a *concept for applying model-driven engineering* to these scenarios and collect *requirements for an MDE infrastructure* to realise the concept. So, we directly address the defined problem statements which are defined in the introduction of this work (see Chapter 1). During the requirements elicitation we also include the expectations of domain experts considering model-driven engineering.

4.1 Interviews with Extension Developers

In this section we present the results of a set of conducted semi-structured expert interviews with 8 industrial practitioners from the Joomla community in 2018. We aimed to *validate the relevance of the previously defined development scenarios* (see Section 2.3) and determine, if practitioners face the described development challenges which can be addressed by an MDE approach. Additionally, we asked the interviewees about their opinion and wishes regarding an MDE approach during extension development. So, *the feedback can be incorporated into the requirements elicitation for an MDE infrastructure*.

4.1.1 Set-up

The quantitative and qualitative answers of the interviewees are categorized based on the following categories:

Experience with Joomla (C1): The first category concludes warm-up questions to determine the experience of the interviewees. During the interview we asked the interviewees about the Joomla versions they worked with and since when.

Development of Joomla/WCMS Extensions (C2): With category 2 we aim to summarize the findings of questions regarding the typical procedure of the interviewees during the development of new Joomla/WCMS extensions. This includes the number of extensions, usual development scenarios and the average time, they require, to develop new extensions. In order to confirm the relevance of the previously described development scenarios (scenario 1 and 2), we also asked questions considering the augmentation of existing extensions. The questions were designed for Joomla components, since they represent the most complex extension kind in Joomla. This should allow a more objective generalisation of the answers.

Migration of Joomla/WCMS Extensions (C3): In category 3 we cluster the results of questions considering the migration of Joomla/WCMS extensions. This includes the number of migrated extensions, preferred migration procedure, the target platform (new major version of the same system or another WCMS), and the average time they invest during migrations of extensions. So, we can investigate the relevance of the migration scenario (scenario 3).

Extension Characteristics (C4): To obtain information of the extension characteristics (category 4), we asked the interviewees, which extension kinds they developed/migrated and how complex they are. In addition, we asked if dependencies between own and core or own and third-party extensions exist. So, we gained further insights into the relevance of the previously described scenarios. Moreover, we surveyed the interviewees about the amount of standard CRUD views in their extensions. Based on our experience that most Joomla components comprise a large amount of standard views with CRUD functionality, this question plays a significant role in order to pursue an MDE approach during WCMS extension development. As shown in Figure 4.1, such views comprise a list to present the entities, a toolbar of buttons to provide CRUD functionality, and a detail page to create or edit an entity.

#	First Name	Last Name	Username	Active	Boss
1	John	Boo	johnny81	<input checked="" type="checkbox"/>	<input type="radio"/>
2	Mary	Brown	missmary	<input checked="" type="checkbox"/>	<input checked="" type="radio"/>
3	James	Mooray	jijames	<input type="checkbox"/>	<input type="radio"/>

Create
Edit
Delete

First Name Active ☒

Last Name Boss ☒

Username

Figure 4.1: Standard View with CRUD Functionality within a WCMS Extensions

In a pre-study, we investigated the amount of views in regard to their type (standard CRUD/custom) in actual extension projects. We applied a structured process to obtain unique extensions with components and available source code on the official Joomla extension directory [170]. Thus, the most extensions represent data-intensive applications by their own. We adjusted the filter criteria to only search for extensions that are *components* for *Joomla 3* and *free to download*, what resulted in 751 extensions. With a website crawler we were able to download 92 extension archives directly. Every other download link referred to an individual download page which we could not automatically address. However, we also added the top 50 extensions (overall user rating) manually to our extension set. We removed duplicates, extensions that were tagged as component but were none, and extensions that were not installable or led to errors. In total, we considered 50 extensions with 592 views and found that 212 views were standard list views and 191 were standard detail views. In short, 68.07% of the views we inspected are standard views with CRUD functionality.

MDE Approach during Extension Development/Migration (C5): Category 5 concludes questions regarding the challenges during extension development and migration. We asked for potential pitfalls during the development process to identify challenges that can be addressed by an MDE approach. Furthermore, we surveyed the opinion of the interviewees concerning MDE of WCMS extensions. We asked for potential scenarios which could be supported by a model-driven approach and inadequate scenarios for MDE. Additionally, we asked for expected features considering an MDE infrastructure. The latter were considered during the requirements elicitation for a suitable MDE infrastructure (see Section 4.3 below).

These categories are designed as abstract as possible to be suitable during further interview iterations, regardless of the preferred WCMS of the interviewee. So, we address the external validity threat which is based on the fact, that all interviewed practitioners prefer Joomla as target WCMS.

With one exception, all interviews were recorded and transcribed. In average, an interview took 20 minutes which led to a total time of around 3 hours of conversations, transcribed to 11050 words. Quotations from the transcripts are given in italicized and quotation marks. Citations are presented in a verbatim form from the transcripts. Citations that were translated to the English language are marked with a capitalized T as index.

4.1.2 Results

In this section we present the interview results, grouped by the previously described categories. For each category, we summarize the results in respective tables and present striking answers. The transcribed interviews can be found in Appendix A.

Table 4.1 indicates, that all interviewed participants used Joomla for multiple years. Four of them stated, that they use and develop extensions for Joomla since the first released Joomla version. All but one of the interviewees used and developed extensions for several major releases of Joomla.

Table 4.1: Experience with Joomla (C1)

I	Years of Joomla Experience	Joomla Versions
I1	6	2.5 - 3.8
I2	5	2.5 - 3.8
I3	13	1.0 - 3.8
I4	13	1.0 - 3.8
I5	13	1.0 - 3.8
I6	13	1.0 - 3.8
I7	4	2.5 - 3.8
I8	10	1.5 - 3.8

Addressing the development of Joomla extensions, Table 4.2 summarizes the answers from the interviewees. Half of the interviewed practitioners developed one or two extensions, whereas the others developed between 10 and 100 extensions including components, modules, and plugins. All interviewees typically follow either a clone-and-own procedure or use a boilerplate generator for the development of new extensions. They stated: *"Copy&Paste and adapt what is necessary"*_T, *"We also have now our own boilerplate, we use just to create a component ..."*, and *"If I need to start a new component, I now would start with the boilerplate ..."*.

In reply to the question of how much time they spent to develop an extension, the interviewees stated to require 2 to 8 hours to have an independent installable component with no business logic following their usual (mostly clone-and-own) procedure: *"... takes us, I think, two hours to*

Table 4.2: Development of Joomla/WCMS Extensions (C2)

I	# of Extensions Developed	Procedure to Implement a new Extension	Average Duration	Augmented a component by
I1	1	Clone-and-Own	a couple of hours	Modules
I2	3	Boilerplate	When we use our boilerplate it still takes us, I think, two hours to set up everything. So, two hours for a component.	New views, Modules
I3	2	Clone-and-Own	Two hours for a small Plugin	I wanted to ... create clean interfaces
I4	10-15	Boilerplate	Having a backend view and a front- end view it's between four to eight hours ... in an hour I can make a frontend and backend view, but that's a very basic.	New views
I5	100	Clone-and-Own	Between 2-4 hours and 100 years	New views, Plugins
I6	20	Clone-and-Own	n.a.	Modules
I7	2	Clone-and-Own	3 to 4 days	New views
I8	40	Clone-and-Own	10 minutes	New views, Modules, Plugins

set up everything. So, two hours for a component". Another interviewee said: *"... it's between four to eight hours and that of course is still quite a bit more work..."* and *"... in an hour I can make a frontend view and backend view, but that's a very basic"*. One interviewee stated that he requires 3 to 4 days for an installable and tested component.

Additionally, they stated that they also augmented existing components by new views, modules, and plugins. This includes both own and third-party components, e.g. by own modules. Considering the typical scenario of augmenting an existing component by a new view even the interviewee with no experience in this scenario sees the relevance of it: *"... it would be a nice case, but not one I use. But it's a use case!"*. Typically, the practitioners' procedure during augmentation of existing components is copy and paste of existing code. They stated: *"... copy and paste and adapt", "Copy an old one and change what needs to be changed", and "Copy an existing view, copy the existing model, change the names, change the database queries where needed. ... in the template file ... change the column names. ... create the filter if there's a filter. ... copy the table files"*. However, one interviewee answered that *"For modules we use simple module creator."*

In Table 4.3 we conclude the answers to the questions considering extension migration. 6 interviewees migrated extensions from one major Joomla version to another one. One interviewee stated, that he also migrated extensions from WordPress to Joomla. Three of the respondents migrated between 10 and 20 extensions throughout their life cycle to different major Joomla versions. The common procedure during extension migration is to *"Rewrite and fix errors until it works"*. This also includes to rewrite some parts or the whole extension anew, following the new API conventions.

Table 4.3: Migration of Joomla/WCMS Extensions (C3)

I	# of Extensions Migrated	Procedure to Migrate a Component	Average Duration	Target Platform
I1	2	n.a.	"a couple days" (8 h/day)	New Version
I2	-	-	-	-
I3	-	-	-	-
I4	10-15	"Rewrite and fix errors until it works"	"a couple of months" (8 h/day)	New Version
I5	15-20	"Rewrite most of the extensions to get rid of old stuff." _T	9 months (10-15 h/week)	New Version Other WCMS
I6	n.a.	"Iterative until it works." _T	"Within 2 years" _T	New Version
I7	2	"Iterative until it works." _T	one year (40 h/week)	New Version
I8	15	"Read a migration guide (if it exists) and then fix errors until it works." _T	6 months	New Version

With regard to their migration procedures, the practitioners reported that it requires them a couple of days for an extension migration to a new release version: "... migrating to the next version took maybe a couple days." If the number of extensions and complexity grows, it took them months, as some interviewees stated: "You should take a year. I think so if you want to do it well. Of course, taking a year is not that I'm working full time for one year to migrate the extension because the migration happens next to the other work I need to do. The daily business goes on. ... Maybe without any interruptions, it still will take a couple of months" and "It took me 6 months to migrate 10 extensions".

In consideration of the extension characteristics (C4), the answers are concluded in Table 4.4. One interviewee only dealt with plugins, whereas another interviewee implemented and migrated only modules. All the other interviewees developed and maintained various extension types, while 4 of them worked with all main extension types (components, modules, plugins).

We asked the interviewees for an estimation of their extensions' complexity. All of them stated that their extensions are complex in consideration of code amount, individuality, and dependencies to other extensions. I2 stated that their components are quite complex: "Yeah, they're complex. While we have one really large one we're working on. It's called Image Manager ... it's only a backend component just to organize the images and it has only one view. ... it has some extra models that so they popping up with some extra information but it's only actually one view. ... high level of individuality ... compared to standard list views."

I4 answered: "I think my one component ... is a pretty complex one, because if you look at, for example, there's a view there called template in which you put all your settings you only use for import or export. And depending on some settings you have it's pulling different XML files and rendering them individually. That is not something I really see in being done in Joomla. So in that sense, I think that view specifically is very complex. ... with the template view there's actually an extra toolbar button to switch between the basic mode and advanced mode. You have test buttons for testing FTP connections, testing file locations, and other things like that. Well, that's not something you have in your general edit screen. Because most edit screens are simply for data input and this is doing more than that."

Table 4.4: Extension Characteristics (C4)

I	Extension Types	Amount of CRUD Views	Dependencies between Extensions
I1	Modules	n.a.	Module depends on component data
I2	Components, Modules, Plugins	less standard views	Module depends on component data, Plugins manage data of component
I3	Plugins	n.a.	Plugin depends on third-party component
I4	Components, Modules, Plugins, Libraries	90%	"I'm using the model from the third-party component that has the logic in it"
I5	Components, Modules, Plugins	100%	Component checks, if plugins are installed ... show plugin-specific features
I6	Modules, Plugins	80-90%	Modules depend on component data
I7	Components, Modules	60-65%	Modules depend on component data, Component depends on plugin and core component (user component)
I8	Components, Modules, Plugins, Libraries	100%	Modules depend on component data

The answers of I5 and I6 were more quantitative: *"My largest component was a user management component with 33 views in the backend and 7-8 views in the frontend. Later I separated the component into 5,6,7 components with 5-6 components each and dependencies to each other"*_T and *"... in the backend we have 15 views and 7 in the frontend"*_T.

In accordance to our findings concerning the amount of standard views with CRUD functionality, they stated: *"I think that ninety percent, at least, are standard list views."* Other interviewees stated around the same amount. They think that around 60 up to 100 percent are standard views: *"Most views are pretty standard because Joomla is based on listing pages and edit pages"* and *"... backend was almost standard with list and form views. Only the details were more individual"*_T.

With respect to dependencies between their extensions, all interviewees stated that their extensions are concerned. I2 stated: *"Yes, we now have a component with five views and also a module and also a couple of plugins. It's a music database where teachers can collect songs they can actually do with the children in the class. So, they open the player, in the player there are all the songs and they can click on the songs and the player will start playing and they have all the song texts next to the player. So that's a really complex application with a lot of fun of use and also a module which is also a player and a lot of plugins, which are actually doing all kinds of stuff in the backend"*, whereas I5 answered that *"... a component detected if plugins or other components were installed to display specific features or menu items based on them"*_T, and I7 told us: *"... component to system plugin, content and profile module to component, ... component to the core user component"*_T.

Additionally, dependencies to third-party extensions exist. I3 *"... tried to get my hands dirty on a REST component"*, whereas I4 stated: *"I have done that. I'm still doing it. The example here's in my extension I'm exporting prices from a webshop and the whole logic of price calculation - I don't want to recover that - so I'm using the model from the third-party component that has the logic in it."* I5 has a loose coupling between own and third-party extensions: *"If a third-party extension and the plugin is installed, an additional field is displayed. If not then not."*_T

Table 4.5: MDE Approach during Extension Development/Migration (C5.1)

I	Challenges during Extension Development	Potential Scenarios for MDE Support
I1	Adhere to Joomla conventions	Initial and further development
I2	n.a.	Initial development
I3	n.a.	Initial and further development of dependent and independent extensions, migration of legacy extensions
I4	Adhere to Joomla conventions	Initial and further development (including migration to new platform version)
I5	Keep up with the latest standards	Initial and further development of dependent and independent extensions, support during deployment of new extension version
I6	Adhere to Joomla conventions	Initial and further development of dependent and independent extensions, initial migration step, augmentation of existing extension by new dependent ones
I7	Understanding of the underlying API	Initial development and migration to new major platform version
I8	Adhere to Joomla conventions	Initial development and migration to new major platform version

Considering the last category (C5), which is relevant for this work, we asked developers for common challenges occurring during extension development. Additionally, we asked them for their opinion on model-driven approaches during development and where they see potential scenarios (see Table 4.5). Furthermore, we asked the interviewees what they expect from an MDE infrastructure in this context (see Table 4.6). Responses to these questions are presented in the following paragraphs.

Ensuring the required code conventions and using the API of the underlying system is one of the most common challenges as stated by the interviewees. I1 stated *"There are always mistakes."* as answer to the question, if adhering to the Joomla conventions leads to errors. I8 gave the following answer: *"Where I had big problems, was with language constants in the manifest file. There is a filename, the component name and another name in the XML and a description. Especially in the plugins. One file had to be similar to another file and then it had to be in the language file or not ... it was very confusing"*_T and *"These conventions, ... some of the conventions are code style. I take that from PHPStorm ... but then PHPStorm wasn't one hundred percent in line with Joomla. ... It would be nice if they decide once and for all and then leave it"*_T.

As answer to the question, if such errors are common, I4 stated *"Yes, renaming classes after copy&pasting a model is standard procedure almost. ... if you are doing a couple of views I would say it's like a hundred percent chance that you forget something somewhere. Because you have to change the name of the model you have to change the name of the controller the name of the view, change the name of the table file. And the other thing you forget is change the query in the table that needs to be queried. That is something also forget. It's a tedious procedure."* I7 stated that the biggest challenge is to understand *"... the MVC pattern and how it's implemented by Joomla and the whole Joomla framework in general."*_T I5 stated, that the biggest challenge is to keep up with the latest standards of the API.

We asked the interviewees, if they are open for a model-driven approach during extension development and in which scenarios they would apply MDE. Some of the interviewees already use a kind of code generator or MDE infrastructure like I1: *I've played with the Component Creator, but what the component creator does, is take away a lot of typing. Which is fine, but what it does not do is, it's difficult to incorporate it in the workflow. So, if you use Component Creator to start a component it's fine and if this component is just implemented and that's it, great. But if you want to take these components and maintain it or to fill it up further or give it more possibilities or functionality, I think the component creator is not really helping. It's only the first bit. But my idea was that if it's only helping me in the first bits of typing a lot of stuff to save a lot of time. I can also automate this otherwise.*" I2 uses a boilerplate generator in the initial development phase: *"Sometimes you need a boilerplate to create everything. It could actually help and ... I think this is really useful to speed up the process and actually when you have to create a standard component which has to do something really easy you can make one really quick. So it can be a time saver. Yes I'm sure it can be."*

In the context of a textual modelling language for WCMS extensions, I4 stated that: *"... if I add a new view in the model file, I would still copy&paste a view from there because the structures mostly going to be the same. But there's less to change because there's only one single file where I need to change maybe two names or three names then the rest will be generated. So, it's less error prone than what we're doing now."* Regarding suitable MDE scenarios, I4 emphasized the need of generation for different major platform versions: *"... because otherwise, I don't need a code generator. I can just change everything manually again. But having the main part of my logic in the code generator would allow me to just export that."*

The answer of I5 was based on experiences with the Laravel framework which follows a kind of MDE approach during initial development: *"I built something with Laravel sometime and they have such a generator inside and in principle I think the idea is quite elegant."*_T Answering the question of if a generator approach would be suitable for extension development I5 stated: *The problem is ... that they are never up to date. That's what I think is the biggest problem with the generators, so I think the idea is chic but I don't use it myself for the reason, that when I use a generator I want to use an all-in-one solution. Where I don't have to check afterwards if it contains the latest changes from two weeks ago. That prevents me from it. By copying where I have to see through, I don't lose much more time."*_T

Considering the scenarios we presented above, I5 added the following example: *"... build something, you have a standard structure: table class, view, controller, model and now comes the requirement: I need a new field. I have a system with thousands of data already and now I need a new field in the database. Where in the worst case is a foreign key on something else. With something like this it would be nice to get some support."*_T

I3 gave positive feedback on the previously described scenarios (see Section 2.3) whereas I6 emphasized scenario 2 and 3, the initial migration step and the augmentation of existing extension by new dependent ones (new Joomla module for existing Joomla component). I7 told us that the initial development scenario (scenario 1) is suitable for an MDE approach: *"In general, I think it's an advantage to be able to generate something. ... it is always an advantage if you can generate the basic scaffold. Then you have a consistent code quality, everything works and can be installed. Then you can build on it."*_T Addressing the migration scenario, I7 said: *"Manual changes are always expensive. It would be nice, if you could generate it. That's what I would have wished during the migration of our components."*_T. Similar answers were given by I8: *"... for new projects or the migration of existing extensions."*_T Additionally, I8 stated that an MDE approach would not be suitable for the development and migration of individual features.

Addressing the expectations and requirements for an MDE infrastructure for WCMS extension development, the following answers were collected (see Table 4.6).

Table 4.6: MDE Approach during Extension Development/Migration (C5.2)

I	MDE Infrastructure Expectations
I1	Integrability into continuous development process, history for generated artefacts
I2	Model creation dialogue (wizard)
I3	Custom code integration
I4	Generators for different platforms
I5	Generation of standard CRUD views, up-to-date generators, model creation dialogue (wizard), core support
I6	Optional model creation dialogue (wizard)
I7	Partial code generation
I8	Generation of standard CRUD views

The generation of standard CRUD views is an emerging requirement for a model-driven approach as I5 and I8 emphasize: *"For standards like forms and lists you would not have to write any code at all"*_T and *"Handling of the entities. ... Including foreign keys and constraints. These CRUD views should also be easy to handle"*_T.

I7 added the need for partial code generation (scenario 4): *"Easily add new views to an existing component."*_T, whereas I4 stresses the importance of generators for different platforms supporting scenario 3 and 5 (migration and modernization of legacy extensions): *"What I would expect is that if I have my logic inside the code generator it would spit out a component in the new style that I put in a different engine and the engine gives me different code to be doing with Joomla 3, Joomla 4, or whatever platform it's supposed to be running on."*

Another expectation is stated by I5 considering the core support during extension creation: *"I need the access management of Joomla ..."*_T

Some interviewees highlighted the need for modelling support, e.g. by a model creation dialogue (wizard). I2 said: *"Just a simple wizard that you have to just type in and it will make the whole model for you and I think that just could speed up the process even more. ... just fill in the name of the component and then you want a module created and a plug in or just a component and you have to take which one you need and the model knows exactly what kind of code to generate."*

I4 stated: *"It would be nice if I do copy&paste, it would ask me what's the new name of your view and then because the names are always consistent it asks me for the name and could consistently change it for me"*, whereas I5 required: *"You build yourself a basic structure, so to speak, say ok here, ... I have these table fields and at the end I want to output it, preferably mapped to Joomla. That I can say ok I have a list view and I can also tell this table to take the data. ... With commands like: build view x,y,z ... in any form a wizard"*_T.

I8 concludes: *"That you can define what kind of module you want ... just a few templates and how the module would handle the data. Also, I would expect different templates for plugins."*_T

One interviewee (I3) indicates the need for custom code integration: *"... enable the generator to hook into ... with your own custom code or with template kind of things that ... you're not forced to hack into the generated code but to just have enough possibilities to do your own stuff."*

I1 emphasizes the integration of the MDE tools into a continuous development process: *"I expect that the generator is not a generator once and change never option. I expect that it's meant to be part of a continuous developing situation. ... If it appears to be a bug in my component after six*

months, I want to be able to go back to the last one that was generated or the last one before that or something like that. ... or at least have a history, yes. Because often if you find a bug, then it can be a totally new bug. It also can be a known bug, that you had before that has returned for some reason." In this context, I5 stated: *"The main requirement is that the infrastructure is up-to-date. ... it should write as much code as possible. And this should not be limited to the initial development, it must also be forward compatible. So, with one click my component will be updated."*_T.

4.1.3 Interpretation

In the previous section, we presented the interview results with industrial practitioners. All of them had many years of experience in extension development and migration for Joomla and other WCMSs (C1). This indicates, that the interviewees are experts in extension development and gave reliable answers.

Although some developers use boilerplate generators to create new extensions (C2), most of them follow the clone-and-own approach, using existing code as reference for new code. Following this approach, the development takes at least hours - even for a simple installable extension. This also applies to the development of depending extensions like modules or new features for existing extensions like new component views. This confirms the challenges during initial extension development as described in Section 2.2.2.

Considering the migration of extensions (C3), it takes the developers at least a couple of days to migrate their extensions to a new Joomla version. For the migration of more complex extensions, however, they require months. This challenge is in line with our finding as described in Section 2.2.2. An interesting finding of the results is, that no interviewee uses a tool for extension migration. The typical migration approach is mostly *"... fix errors until it works."* I5 even rewrites his extensions completely. Most of the interviewees typically migrate their extensions between major versions of the same platform. However, one interviewee also migrated to another WCMS. This scenario corresponds to our defined scenario 3 and can be directly addressed by an MDE approach, using a platform-independent modelling language for extension description and platform-specific code generators.

Considering the amount and types of his extensions, I5 puts high effort into the migration, even though all component and module views are standard views with CRUD functionality (C4). This statement was confirmed by almost every other interviewee for their extensions. The practitioners typically put tremendous effort into the migration of schematically recurring extension code. Moreover, the typical extension characteristics (C4) support our own investigation of schematically recurring code in WCMS extensions. As the results show, the amount of standard views with CRUD functionality is so high that most of the development and migration steps can be performed automatically. Another indicator for the appropriateness of MDE are the existing dependencies in the developed extensions. All interviewees have implemented extensions with dependencies to other custom extensions (own or third-party). This is consistent to our described scenario 2 and could lead to further challenges based on extension evolution. The migration of a dependency could necessitate a migration of depending extensions.

As the results show, adhering to the WCMS's conventions, like coding standards and API requirements, is the main challenge during extension development (C5). This challenge can be addressed by a code generator since they typically comprise the technological details like API function names or naming conventions for files and code artefacts [221]. An interesting finding is that experienced developers are not open for code generators. This finding coincides with the results from other empirical assessments of MDE adoption in industry such as [95] and [142]. As our interviews reveal, this is based on the developers' experience with existing tools like boilerplate generators, which are rarely up-to-date or not integrable into their development process

(e.g. for extension evolution). However, all interviewees see potential for MDE support during the scenarios which are described in Section 2.3, provided that certain requirements are fulfilled by the MDE environment. Besides the generation of schematically recurring code fragments (complete extension and partial code generation) for different WCMS platforms, the integration into a continuous development process is required. The code generators must be kept up-to-date and should support custom code. In addition, wizards for model creation are required by some interviewees. Using such wizards could lower the hurdle for developers with little experience in using an MDE infrastructure.

We can conclude that, considering RQ1.1, the relevance of our predefined scenarios is strongly confirmed by the industrial practitioners' statements based on the interview results. During the last decade the interviewees had to face similar recurring challenges, such as implementing requirements with a high amount of schematically redundant code and depending extensions. Additionally, they had to go through major platform changes with the corresponding migrations of their extensions. In both cases, extension maintenance requires large effort. With regard to RQ1.2, the previously described scenarios can be directly addressed by MDE, since decreasing development effort for these scenarios is an acknowledged strength of MDE. All interviewed practitioners also see potential for MDE support during the presented scenarios. The collected expectations can directly be used during the elicitation of requirements for an MDE infrastructure for WCMS extensions in order to address RQ 1.3.

4.2 MDE Concept for WCMS Extensions

In this section, we propose concepts for the integration of a model-driven approach during the previously described development scenarios of WCMS extensions (see Section 2.3). So, we aim to answer the question of how MDE can support developers during these scenarios (RQ1.2). The relevance of these scenarios is confirmed by industrial practitioners (see Section 4.1). To apply a model-driven approach to a new domain requires the implementation of a set of MDE artefacts (MDE infrastructure). To this end, we define concepts for integrating modelling languages and models as well as model transformation tools into the development process of WCMS extensions.

In the following subsections, we distinguish between independent and dependent extensions in accordance to the previously described development scenarios (1-3). This includes the initial development and the evolution of both independent and dependent extension. Additionally, we enclose the related sub-scenarios 4 (partial augmentation) and 5 (modernization of legacy extensions). So, in order to ensure the maintenance of generated artefacts, techniques for further development will be examined. In this context we consider (re-)development by hand and the maintenance by using an MDE infrastructure. Figure 4.2 illustrates an overview of MDE infrastructure support during the development of WCMS extensions.

As previously described, the main infrastructure artefact is represented by a modelling language for WCMS extensions. Based on this language, model instances for actual extensions, e.g. for conference management, can be defined. Since these models represent the main artefacts within MDE, the language must consist of the necessary features of the domain, in our case WCMS extension features (see Section 2.1). In addition to an appropriate modelling language, the infrastructure must comprise facilities for code generation. To this end, various approaches for code generation exist (see Section 3.3.2). In this work, we focus on template-based code generation. The code generation step must result in installable WCMS extension packages which are directly deployable to a running WCMS instance such as a Joomla or WordPress site.

In order to support the augmentation and re-engineering of existing extensions, we will also consider legacy extensions within our MDE concepts. According to the definition of Seacord et al. in [209], legacy code is a popular indicator in the maintenance context and often associated with old code that is no longer needed. Demeyer et al., however, describe legacy code as "valuable

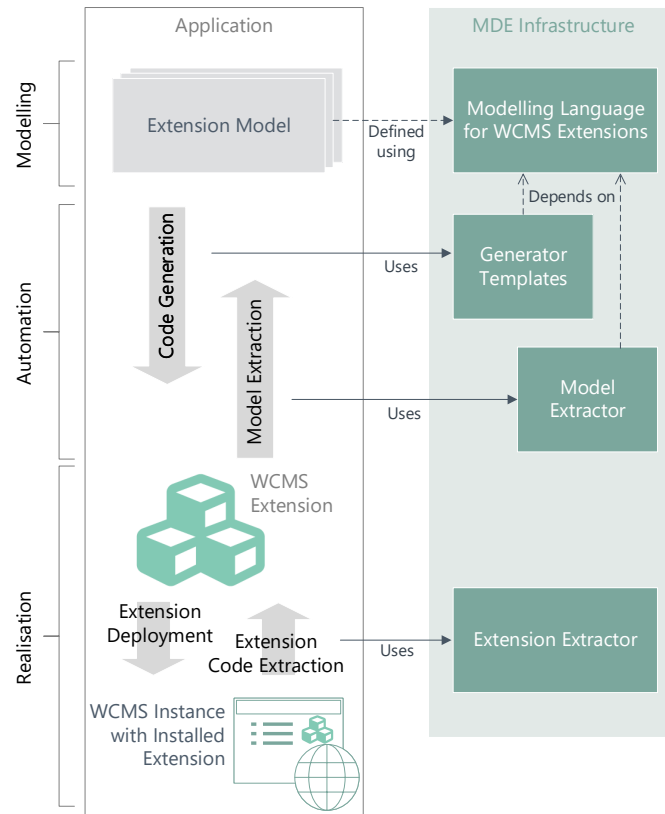


Figure 4.2: MDE Infrastructure Concept for WCMS Extension Development (cf. [28])

software that you have inherited" in [54]. In [253, p.28 f], the author defines criteria for a more accurate classification of legacy code based on basic (axiomatic) and derived criteria. This criteria is based on the established definitions of Fowler et al. in [70], Feathers in [64], and Martin et al. in [146]. In the context of WCMS extensions, two kinds of legacy extensions exist. The first kind of legacy extensions is represented by *installable packages which can be found, e.g. in extension directories*. These extension packages could be used directly for model extraction as part of a reverse engineering step (cf. C2M in Section 3.3.2). By providing a model extraction tool, this step can be automated. Another kind of legacy extensions are *already deployed extensions within a running WCMS instance*. In WordPress and Drupal, both kinds are the same, since the deployed extension structure is the same as the one of the installation package. Though, in Joomla the structure of a deployed extension differs, since the extension files are moved during the installation routine. Therefore, a code extraction from a running instance must be performed in order to obtain an installable extension package.

With these infrastructure artefacts various development directions are possible. Besides forward engineering, reverse engineering of WCMS extensions is supported. During reverse engineering steps, we consider model discovery, understanding, refactoring, and augmentation [28] in the following concept descriptions. A combination of both development directions can be used as part of a round-trip engineering process [54]. However, round-trip engineering typically considers changes on model-level as well as changes of the actual code. Though, the intention of MDE is to deal with models as primary artefact without the need of custom code refinements [221]. Necessary custom code typically represents individual features, which are not considered by the utilized modelling language. So, if the infrastructure does not consist of suitable mechanisms for custom code, round-trip engineering is rather exceptional. An exception are approaches based

on framework-specific languages which are intended to complete existing framework features by an MDE approach as explained by Antkiewicz et al. in [7], [6], and [8]. In this work, we limit our research to the forward and reverse engineering of WCMS extensions and will not research possible round-trip engineering scenarios.

As Norrie et al. describe in [160], there is little recent research on the development practices for WCMSs. Most of the existing approaches with similarities to extension development such as presented in [204, 205], [235], and [25], present platform-independent meta models and generators for the development of specific WCMS instances and their extensions. Though, alternatives to extension development is not extensively researched. Only the work presented in [66] has similar intentions providing an MDE approach for the development of extensions for the Dot-NetNuke WCMS. As this WCMS has a limited extension mechanism, another language, providing suitable abstractions and automation facilities for a more sophisticated extension mechanism is needed. Existing platform-specific code generators for WCMS extensions are limited to initial development scenarios for independent and corresponding dependent extensions but do not provide partial extension development or support extension developers during extension migration or modernization activities. In Section 5.1 and Section 6.1, we present further related work to extension development and MDE in this context. Following, we present MDE concepts for the elicited development scenarios under consideration of the domain-specific roles and required infrastructure artefacts.

4.2.1 Model-Driven Engineering of Independent Extensions

During the development of independent extensions we distinguish between initial development and iterative (re-)development (see Section 2.3.1). In order to develop independent extension in a model-driven manner, both scenarios must be supported by at least a modelling language and suitable code generators, following a forward engineering approach (cf. [188, 186]). Figure 4.3 illustrates the proposed tool support for extension developers during MDE of independent WCMS extensions and shows the involved roles and artefacts.

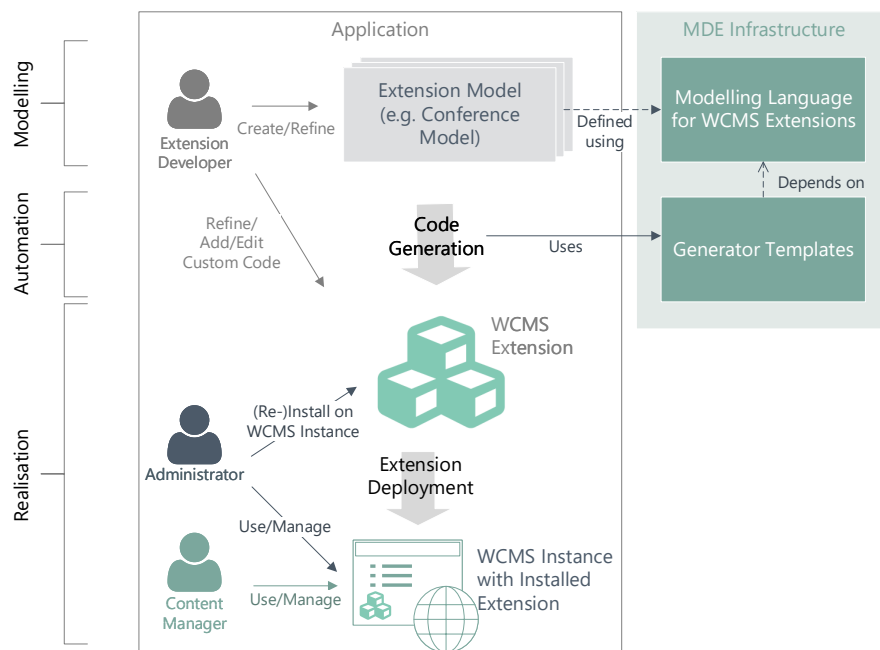


Figure 4.3: Model-Driven Development of Independent Extensions

The initial development, which is already supported by exiting boilerplate generators (cf. Section 6.1), must result in an installable WCMS extension package. This extension package must adhere to the WCMS-specific conventions on file and code base (cf. problem statement 1). Using an MDE infrastructure for initial development requires the specification of the desired extension features within extension models by extension developers. Model information can then be used by WCMS-specific code generators. These merge generic code fragments with features from the models during code generation. Through the formal platform-independent description of extension features and the use of platform-specific code generators, different platforms can be addressed concurrently. This allows the code generation for both extensions for different platforms, such as Joomla and WordPress, and extensions for different versions of the same system.

Generated extension packages can be installed by WCMS administrators to a running WCMS-based website where they can be used and managed by administrators and content managers, depending on the respective extension's remit. Iterative refinements like add, edit, and delete operations must be done on modelling level to ensure up-to-date models which will be used during a re-generation of the WCMS extension. The result, a new extension version, can then be (re-) installed on WCMS instances. If an older version of the extension is already installed, the respective files will be overwritten and, if required, respective database tables have to be updated. This requires the generation of adequate database update scripts which are processed during the re-installation process. This sub-scenario is not covered by existing boilerplate generators (problem statement 5) as we conclude in Section 6.1.

Refinements to the generated code like the implementation of custom code requires no further action, if a further development is not planned. In this case, the use of existing boilerplate generators may be sufficient. Though, if a developed extension has to be maintained in a model-driven manner, any refinement and additionally added custom code must be considered during each iteration. To this end, an adequate approach like code models, model annotations, or protected regions must be provided by the MDE infrastructure (cf. Section 3.3.5).

4.2.2 Model-Driven Engineering of Dependent Extensions

As previously described, the development of dependent extensions is not adequately covered by existing related work, e.g by code generators in the domain. Therefore, we propose a concept for model-driven engineering of dependent extensions to reduce effort ensuring high quality in WCMS extensions (cf. problem statement 1). Moreover, our concept aims to reduce the high effort which is required during reverse engineering of existing extensions (cf. problem statement 3) and the maintenance of dependent extensions (cf. problem statement 4). Similar to the concept for MDE of independent extensions, we also consider both initial and iterative development during the model-driven development of dependent WCMS extensions (cf. [186]). This concept addresses the challenges during conventional development of dependent extensions (see Section 2.3.2). Since this scenario is based on interdependencies between extensions, various sub-scenarios must be covered by an MDE concept. This includes the development of extensions which depend on *existing (legacy) extensions*, *extensions which are also developed in a model-driven manner*, at best with the same MDE infrastructure, and *features of the WCMS core platform*. A straightforward example for this scenario is the development of a Joomla module, which illustrates the data of an already existing Joomla component. In Figure 4.4, we incorporate the involved roles and artefacts of our concept for MDE of dependent WCMS extensions. Besides the modelling language and generator templates, reverse engineering facilities for extension and model extraction are considered.

Starting with the initial development of a dependent extension, our MDE approach proposes the creation of an extension model with an adequate modelling language. To specify dependencies to another extension on model level requires a respective extension model. So, the correct identifiers can be used during the code generation process and, once deployed to a running

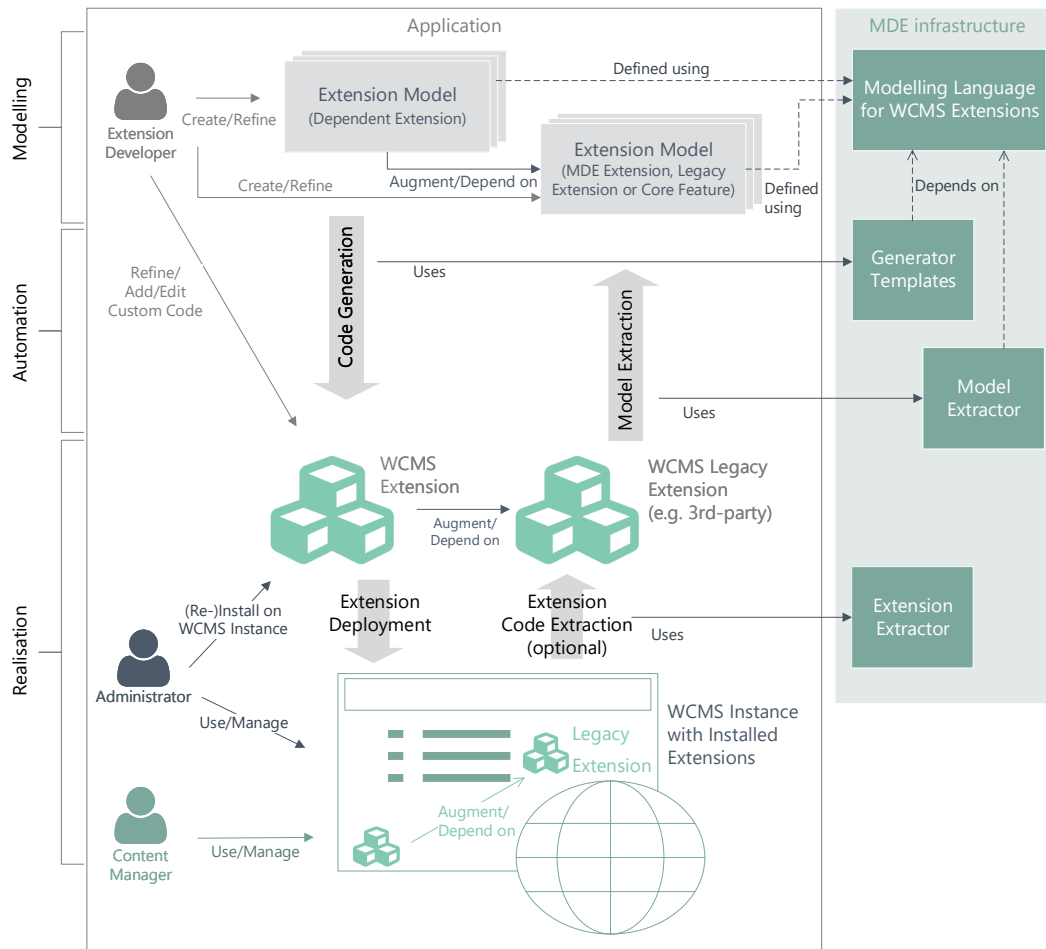


Figure 4.4: Model-Driven Development of Dependent Extensions

WCMS instance, a correct behaviour can be guaranteed. If the extension, to which dependencies exist, is already formally described within an extension model (e.g. by applying MDE during scenario 1), extension developers can follow a forward engineering approach for both dependent and independent extensions.

If no model exists, our concept provides that model information for existing extensions may be extracted within a reverse engineering process. This process can be performed manually by extension developers or automated, supported by a model extraction tool which creates extension models based on existing extension packages. Whereas a manually processed reverse engineering process is tedious for extension developers (cf. problem statement 3), support by means of automation can reduce the effort during this process tremendously. We propose installable extension packages as reference for model discovery, since they comply with the result of a code generation process. Depending on the WCMS, the extension structures of extension packages and deployed extensions may vary, as it is the case for the Joomla WCMS. If the extension package for a required extension is not available, we propose the use of a tool for the automated extension extraction from a running instance on which the extension is deployed. This step however, can also be performed manually and could be unnecessary, e.g. if the extension model is created manually as well.

If both dependent and independent extensions are formally described on model level, extension developers can create dependencies in the dependent extension model, provided that the modelling language provides suitable language elements. In addition, refinements in the model of the independent extension can be made, e.g. if the model was automatically extracted from an existing extension package. Based on the new dependent extension model, the code generator has to create an installable extension package which must homogeneously interact with the existing extension, provided that both the generated dependent extension and the existing extension to which dependencies exist are deployed on the same system.

During iterative development of the dependent extension, developers must ensure to keep the extension model to which dependencies exist up-to-date. In the case, that both dependent and independent extensions are developed in a model-driven manner, every refinement on the dependency extension must be considered in the dependent extension model as well, in order to ensure correct interaction during runtime. Changes (edit/delete) of model elements to which dependencies exist may lead to an unexpected behaviour in the system. Though, this challenge can be addressed easily. If both kinds of extensions are modelled by developers of the same team, every new revision can be detected, e.g. by model versioning.

Similar to the other scenarios, existing custom code must be considered during iterative development. If custom code is added to a generated extension, extension developers must ensure to transfer it to the new extension version. We propose the use of code models or architectural pattern instead of protected regions in order to ensure that generated code is completely overwritten during a re-generation step (see Section 3.3.5).

The detection of changes in existing third-party extensions to which dependencies exist is a particular challenge for extension developers (cf. problem statement 4). Edit and delete operations on critical parts may lead to broken extensions, if new versions of the extensions are deployed to a running instance which also has dependent extensions installed. To address this challenge, extension developers should be in contact with administrators of these running websites. Most systems like WordPress, Joomla, and Drupal include an administration interface for extension update information. If a new version of a third-party extension, to which dependencies exist, is available and ready to install, administrators must inform extension developers to refine the respective extension model and regenerate dependent extensions. The same applies to platform updates, if an extension with dependencies to core features is developed.

4.2.3 Model-Driven Migration of Legacy Extensions

Alternatively to manually performed migrations of WCMS extensions (see Section 2.3.3), we propose a model-driven migration process at a higher abstraction level. This allows the application of common model-driven engineering practices, like, e.g., model refactoring for improving the software quality. So we address the effort which is typically required, if extensions must be migrated due to API changes of the underlying WCMS (cf. problem statement 2). Current tool support does not support extension developers during this process (cf. problem statement 5) and current work in this field does not address the migration of WCMS extensions. Based on common reengineering concepts as described by Demeyer et al. in [54], our proposed migration process consists of three main steps (cf. [238] and [187]):

1. *Reverse engineering* of legacy extensions from an old platform
2. *Model refinement/transformation* (migration/refactoring)
3. *Forward engineering* to a new platform

These steps build the well-known horse-shoe model of (model-driven) software modernization (cf. [202]) which is illustrated in Figure 4.5.

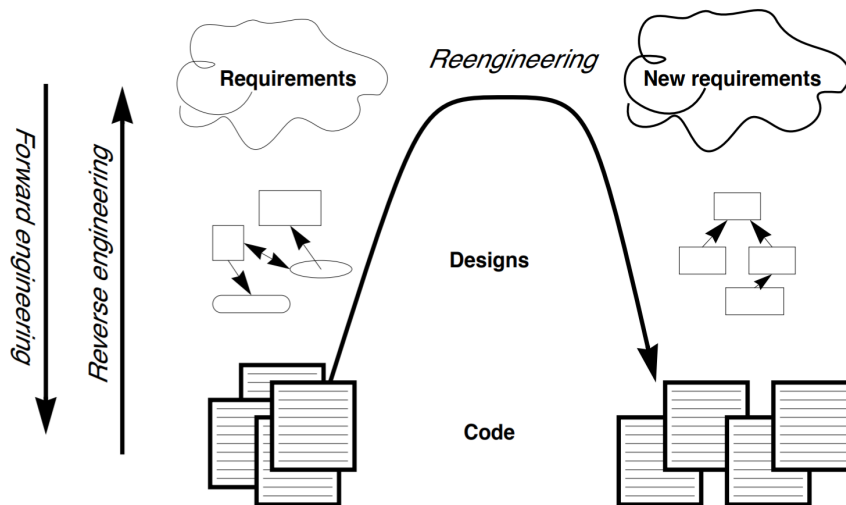


Figure 4.5: Forward, Reverse and Reengineering [54, p. 9]

Similar to the previously described scenario, installable legacy extension packages should be used as input for an automated model extraction. This may require extension extraction from a running WCMS instance, if no installable extension package is available. The resulting extension model should be as complete as possible. In order to keep custom code fragments which cannot be described by the modelling language, we suggest to create code models, which contain platform-specific code fragments and can be bound to abstract extension models. Alternatively, the code can also be incorporated manually after the code generation step for the new platform.

Extension models can be refactored, extended, or migrated to models based on differing modelling languages through model transformation techniques (cf. M2M in Section 3.3.2). These transformations can be performed semi-automatically (e.g. model refactoring), or manually (e.g. model extension). If the language is abstract enough, a discovered model can be used as it is for the forward engineering step.

During a forward engineering step, the extension model should be used as input for code generators which create extensions for the new target platform. As previously described, this can be a platform of another WCMS or a new platform version of the same WCMS.

Our migration concept aims at the migration of extension code. If the data of an already installed extension has to be considered during the migration scenario, additional actions have to be performed. Such data usually exists, if an extension includes an own data management such as Joomla components. Barker proposes the following migration strategy for content in [17] as follows:

1. *Extraction*: Content is extracted from the current environment.
2. *Transformation*: Content is altered, to simply clean it up or to change it to work properly in the new environment.
3. *Reassembly*: Content is aggregated to correctly fit the new environment.
4. *Import*: Content is imported to the new environment.
5. *Resolution*: Links between content objects are identified and resolved.
6. *QA*: Imported content is checked for accuracy.

We suggest to follow these steps as well in order to migrate existing data from already installed extensions. The extraction could be processed automatically to a suitable format, such as XML, JSON, SQL, or another adequate DSL, in order to transform and reassembly it to a refactored extension structure, if required. The result can then be incorporated with the migrated extension as part of the forward engineering process. Alternatively, the data can be directly deployed to a running system which is based on the new platform and has the new extension installed, e.g. by using database scripts.

The proposed model-driven migration strategy is illustrated in Figure 4.6, which shows the roles and artefacts within the horse-shoe model using the proposed MDE infrastructure facilities:

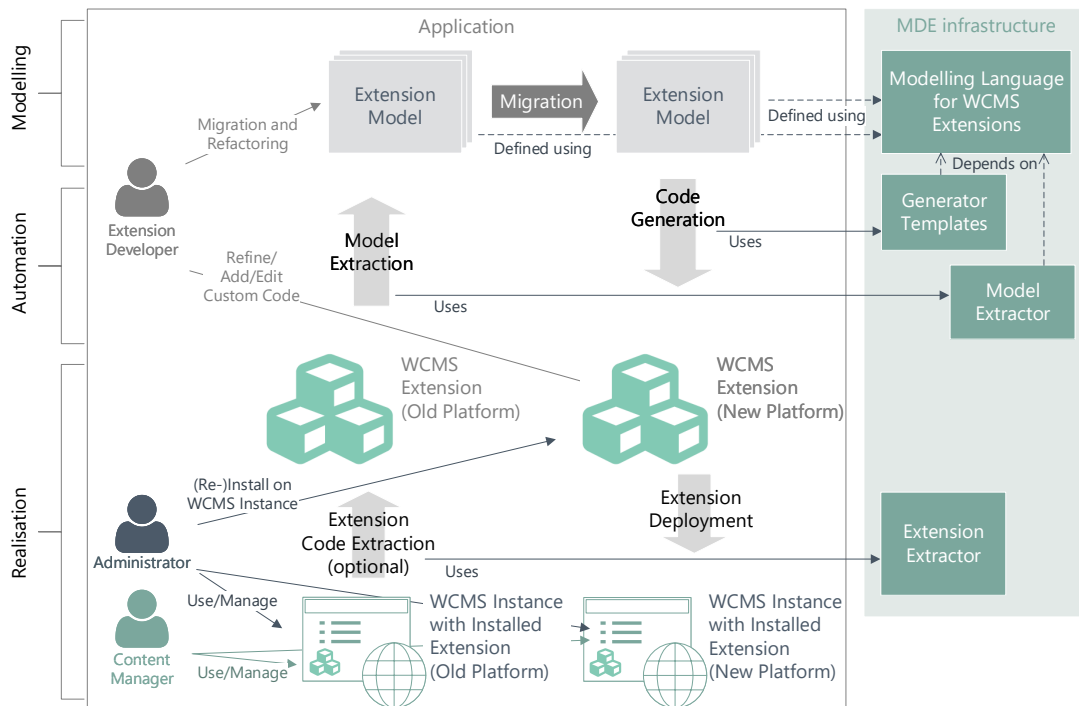


Figure 4.6: Model-Driven Migration of Legacy Extensions

4.2.4 Additional Scenarios

In accordance to the previously presented interview results (Section 4.1), the additional scenarios 4 and 5 (partial augmentation and modernization of legacy extensions), as described in Section 2.3, may occur during extension development. Based on the previously described scenario concepts, including forward and reverse engineering approaches, we address these scenarios by an MDE concept in this sub-section under consideration of the same MDE infrastructure facilities as being used for scenario 1-3.

Model-Driven Augmentation of Legacy Extensions

In order to address the scenario of partial augmentation of existing legacy extensions (see Section 2.3.4), we propose to follow a similar concept as described in Section 4.2.2 (MDE of dependent extensions). This concept, as illustrated in Figure 4.7, is based on an existing legacy extension which is either deployed to a running instance or available as installable extension package. This extension has to be used as input during a model discovery process, e.g. by using model extraction facility. Extension developers can augment resulting models by new features which

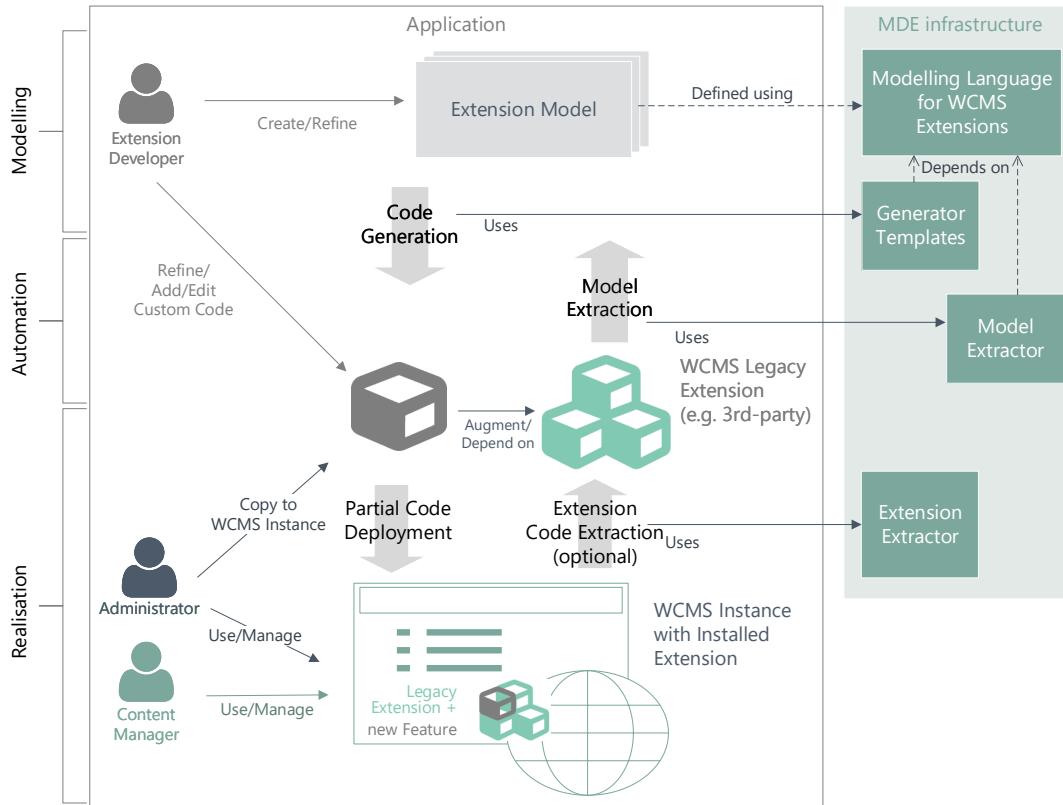


Figure 4.7: Model-Driven Augmentation of Legacy Extensions

augment the existing ones. Typically, developers augment existing extensions by new views (cf. Section 4.1). During a code generation process, a new extension with the new features is generated, based on the input model. Administrators can copy new files from the generated extension package to the original extension package or already deployed extension. This step offers a drawback of the concept, since we limit the augmentation to new files which do not overwrite the original extension code. We propose to use model annotations for original features which are used for references but can be skipped during code generation. If the generator can interpret such annotations during the generation process, original model features can be skipped in order to generate partial code only.

Model-Driven Modernization of Legacy Extensions

Another additional scenario which can be addressed by our proposed MDE concept considers the modernization of legacy extensions (see Section 2.3.4). As previously described, this scenario is relevant, if a legacy extension is an essential part of a running instance, but receives no further maintenance by the original developers. Alternatively, a modernization may be required, if the extension quality has to be enhanced. So, shortcomings in adhered coding standards and architectural guidelines can be addressed. We propose to the same concept as proposed in Section 4.2.3 above (model-driven migration of legacy extensions). For the purpose of performing a model discovery, an installable extension package must be extracted from a running WCMS instance. This extension then is to be used for the extraction of an extension model, which can be optionally refactored on model level. Alternatively, the reverse engineering process can be performed manually by extension developers. In this case, they have to discover and integrate the knowledge of the legacy extension into new extension models from scratch.

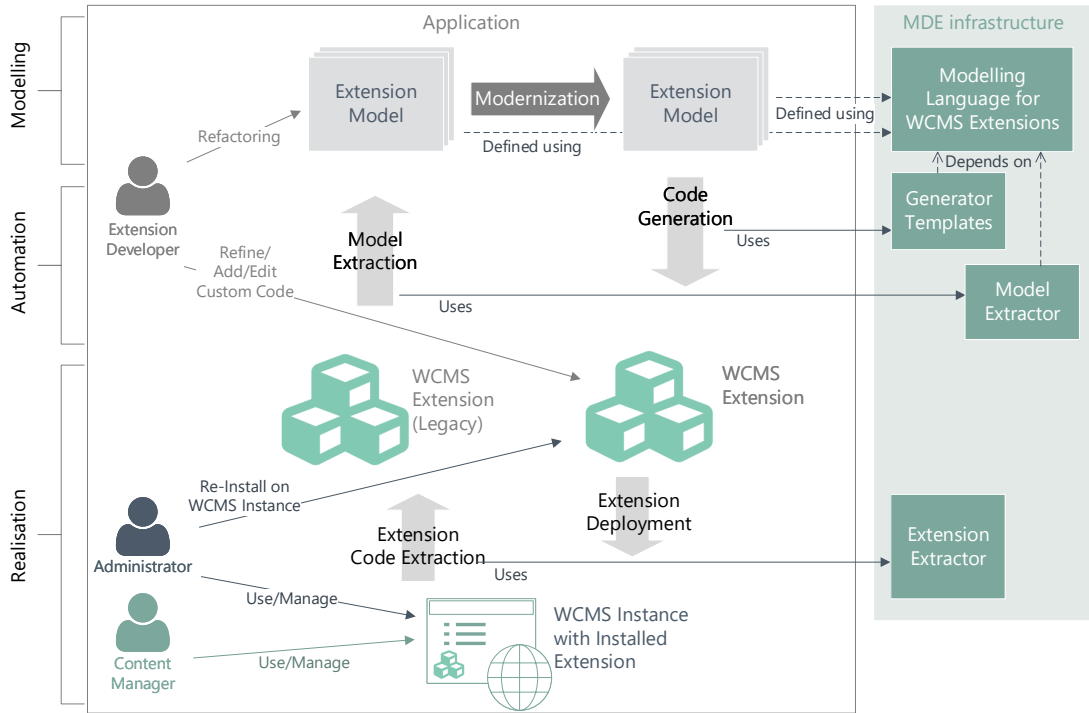


Figure 4.8: Model-Driven Modernization of Legacy Extensions

The modernized extension model is then be used for the generation of a new modernized extension. The difference to the migration scenario is the target platform to which the new extension is intended to be deployed. A migration scenario has the objective of transferring an extension to a new platform, whereas the modernization scenario aims at the modernization of an extension for the same platform. In Figure 4.8, we illustrate the modernization scenario.

4.3 MDE Infrastructure Requirements Elicitation

In order to apply the previously defined MDE concepts, a requirements specification for adequate supporting facilities like DSLs, editors, and code generators, must be elicited (RQ1.3). So, the MDE infrastructure can be composed and evaluated against proper definitions addressing the challenges which are faced by WCMS extension developers (cf. problem statements in Chapter 1). To this end we follow established procedures, as defined in [136], [200], and [178], during requirement engineering of software systems. This mainly includes the execution of *requirements analysis*, *requirements specification*, and *requirements validation*. An essential task during requirements elicitation is the identification of involved stakeholders and their needs in the considered problem domain. In our case the stakeholders are developers of WCMS extensions. To improve the construct validity of our research, we include the results of the expert interviews with industrial practitioners into the requirements analysis phase (see Section 4.1). This decreases possible subjective bias, even though our extensive experience in the domain could be sufficient for the requirements identification.

Additionally, we investigate existing reference extensions to identify the main features which should be supported by an MDE infrastructure for WCMS extensions. As part of our development approach, the list of requirements increases and becomes more accurate with each iteration. We will not specify all infrastructure features completely at the beginning of our development

process. Therefore, we cannot provide a high detailed specification for every possible requirement. However, the main requirements regarding the MDE infrastructure can be elicited at an early stage. Taking into account that they will not change completely, but will be refined in further iterations, the infrastructure components can be composed and developed in an agile manner as described in Section 3.3.6.

Within the following subsections the main MDE infrastructure requirements for the realization of MDE during WCMS extension development will be defined. This includes requirements for a modelling language, model editors, and transformation tools like reverse engineering facilities and code generators. Additionally, we will present overall infrastructure requirements for the whole infrastructure and its integration into the development process of extension developers. The presented requirements set includes functional as well as non-functional requirements. Non-functional requirements¹ typically consider functional correctness, usability, performance, and maintenance of a software artefact [192]. We summarize these aspects to pragmatic requirements to complete our set of requirements.

To facilitate the retrospective validation of the infrastructure, each requirement will consist of a description and acceptance criteria.

4.3.1 Domain-Specific Language

Starting with the main artefact of an MDE infrastructure for WCMS extensions, the requirements for a domain-specific modelling language have to be collected. These requirements must specify the basic needs during the model-driven development of WCMS extensions which will be considered during DSL development.

Lots of guidelines for the development of DSLs exist, whereas the principles of DSLs defined in [28, p. 70] conclude the main non-functional requirements the most appropriate:

- The language must provide good abstractions to the developer, must be intuitive, and make life easier, not harder.
- The language must not depend on one-man expertise for its adoption and usage. Its definition must be shared among people and agreed upon after some evaluation.
- The language must evolve and must be kept updated based on the user and context needs, otherwise it is doomed to die.
- The language must come together with supporting tools and methods, because domain experts care about maximizing their productivity in working in their domain, but are not willing to spend huge amount of time in defining methods and tools for that.
- A good DSL should be opened for extension and closed for modifications, according to the good old open-close principle, stating that "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

Based on these principles the following main functional requirements describe the main aspects which must be considered during DSL development. Since models (DSL instances) are the main artefact of an MDE approach, all of the following requirements address the problem statements 1-4 by lifting extension features to model level.

¹A collection of typical non-functional requirements can be found in the *ISO/IEC 25000* standard. The standard provides a detailed guide considering the quality requirements of software. The standard replaced the *ISO/IEC 9126* standard, which is referred frequently in literature. The standard can be found in [99].

Requirement: Data Modelling

WCMS extensions represent data-intensive applications, i.e. software mainly dealing with a large amount of structured data for a specific purpose [41]. The managed data typically consists of entities with groups of typed attributes and references between them. Therefore, the DSL must allow developers to create individual data models for different domains which represent the required data structure of the extension to be developed. Features like data types, references, and inheritance must be provided by the language to ensure re-usability. Usually, once installed, the data structure of an extension can not be changed or configured within a WCMS instance. Therefore, the DSL must not provide features for a configurable data structure during runtime. However, the data models must be as detailed as possible to ensure the most possible individuality.

Acceptance Criteria:

To fulfil this requirement, the DSL must provide features for a detailed data modelling of a WCMS extension. This includes data modelling features like typed attributes, references between data entities, and inheritance. The data modelling part of the language must not be limited to a specific domain.

Requirement: Data Representation and Interaction Modelling

Data-intensive applications like the most WCMS extensions require user interfaces which allow the management of its data. These interfaces combine the main application logic of the extensions in a form-based way with the possibility to proceed user operations. Usually they provide functionality for operations on data entities (e.g. CRUD) and allow interactions between various representation pages. Interactions can be featured by links from one representation to another. Based on the extension structure of the interviewed developers and our research of existing reference extensions (see Section 4.1.1), WCMS extension representations can vary from *static*, *index* (list of data entity entries), *details* (a specific data entry), *edit*, and *custom* views, whereas custom views incorporate all other representation kinds. Figure 4.9 illustrates an exemplary interaction between different representations of various data entities of a conference management.

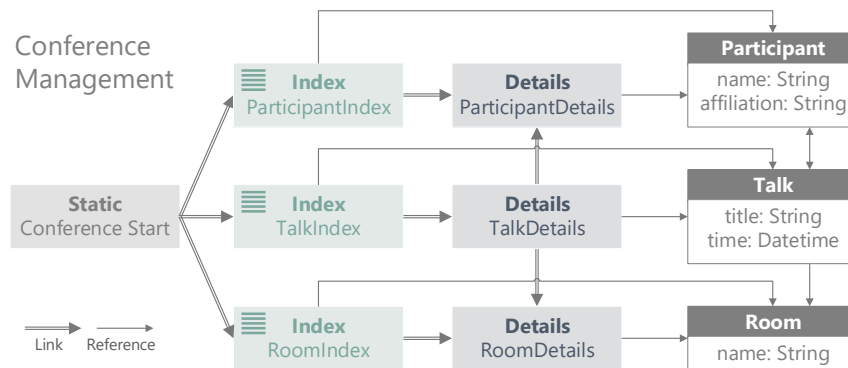


Figure 4.9: Interaction between different representations of various data

A DSL must provide features for data representation and interaction modelling. Thereby, standard operations like CRUD should be hidden by the language within abstract language elements. So, the model can be as small as possible, whereas the feature allows conclusions to the actual extension code, e.g. during further code generation. However, the language should be flexible enough to allow developers to model more individual extension functionality. Extension developers require optional language features to bind individual actions to appropriate view types.

The same applies to the representation specification. The language must provide optional model elements to specify the appearance of a representation. Examples are representations in form of custom HTML forms with sophisticated form fields like e.g. date or colour pickers. So developers can model standard and more individual representations.

The interaction modelling part must allow the specification of a page flow between different representations by a sophisticated link behaviour. This includes both the specification of internal and external links, whereby internal links could contain context information (parameters of HTTP methods) which must be considered during development of the DSL.

Acceptance Criteria:

The acceptance criteria for this requirement is divided into two main part. First, the DSL must provide model features to specify representations of various kinds. For each representation kind, optional language elements for individual representation modelling must exist. Second, the DSL must allow interaction modelling based on these representations. This includes the optional specification of individual actions and the definition of (context) links between several representations in the model. Standard behaviour should be hidden in abstract model elements.

Requirement: Extension Modelling

Besides data, representation, and interaction modelling, extension developers require language features to specify extension-specific meta-data. The minimal requirement to the language are elements for the specification of manifest information and the supported languages of an extension (cf. Section 2.1.3). Additionally, the language must provide a feature to map representations to an extension and if required to a specific site section (frontend/backend) within the extension. So, extension developers can differentiate between the sections of Joomla components or map a representation directly to a Joomla or Drupal module or WordPress plugin.

In addition, a sophisticated language mechanism dealing with different extension kinds must be provided. This includes features for creating references between the extensions. These can be *link, use, or containment relations*. Moreover, the language must be flexible enough to allow further augmentations to special extension features and new extension kinds.

Acceptance Criteria:

This requirement is fulfilled, if the DSL allows developers to create extension specifications with relations to other extensions. The extensions part must provide language elements for meta information, whereby manifest information and supported languages are the main requirements. Additionally, the language should provide a way to include representations within an extension specification. However, the extension modelling part should be abstract enough to hide the technical knowledge of extensions.

Requirement: WCMS Core Support

As described in Section 2.1.3, WCMS extensions can use core features once deployed to a running WCMS instance. For instance, if an extension requires user information, it typically can access an interface of the core to achieve the required data. Therefore, a solution for core support should be provided by the DSL as well. If developers require dependencies between their extension and the core system, the language must provide interfaces to typical core functions. The interfaces should be abstract enough to allow reusing the model for different WCMSs.

Acceptance Criteria:

The requirement can be declared as fulfilled, if the DSL provides platform-independent interfaces to the WCMS core. So, developers should be able to create dependencies which will be resolved when the extension is deployed to a running WCMS.

Requirement: Configurable Generator Hooks

Extension developers require a language feature to highlight model elements which are required during modelling but should not be considered during code generation. So, it should be possible to model parts of legacy extensions to which the new extension will have dependencies during runtime (cf. scenario 2 and 4 in Section 4.2). However, code for the legacy extension should not be generated anew. This feature can directly be used to address the challenge of dependency management between dependent extensions (cf. problem statement 4).

Acceptance Criteria:

This requirement is fulfilled, if the DSL provides a language feature which preserves model elements from being used by the code generator during further transformation processes.

Requirement: Quality Assurance of Extension Models

To ensure consistency within extension models, rules for their well-formedness have to be defined. These model validation rules prevent developers from wrong model specifications such as missing elements, useless references, or inconsistent relations between extensions. Using constraints within automated validation, model violations can be found during the modelling process before the model becomes translated to code. This reduces the defect potential and supports developers to create high quality extensions. Additionally, developers can learn the features of the DSL faster, if they get direct feedback about the model quality.

The definition of language constraints is a popular task during DSL development. Over the past years the Object Constraint Language (OCL) [254] has proven as standard within meta-modelling approaches. The language allows standardized and reusable invariant definitions for modelling languages. These can be incorporated easily into automatic model validation before any model translation. Therefore, OCL constraints should be defined in order to ensure adequate model quality checks.

Acceptance Criteria:

To accept this requirement as fulfilled, language constraints have to be specified. These constraints have to ensure well-formed and consistent models before these get translated to code. In order to ensure reusable constraint specifications, they have to be formally described in an appropriate constraint language like OCL, which may be supported during automatic validation operations.

4.3.2 Model Editors

Besides a DSL for WCMS extensions, developers require a way to create model instances of this language. In order to fulfil this requirement, a set of editors has to be implemented. This addresses the challenge of missing tool support for WCMS extension development, especially for applying an MDE approach (cf. problem statement 5). The following functional requirements consider the main features, extension developers expect from these editors.

Requirement: Textual Model Representation

Model editors must provide a way to create model instances based on a DSL. In the course of this, infrastructure developers have to decide which kind of representation is the most suitable (cf. Section 3.3). This decision determines the required concrete syntax definition. Whereas the language is built on abstract syntax definitions, the concrete syntax describes how the model is represented. For some approaches graphical model representations are the best choice. Examples for this are object-oriented modelling approaches using UML, which provides visual model representations like class diagrams. This allows domain experts to create models without adhering to

any grammar-specific rules. Additionally, visual modelling is more preferred by non-developers. In contrast, textual representations, which follow a certain structure, e.g. provided by a grammar, are more suitable for developers who are used to textual programming. If the textual concrete syntax is close to the natural language, textual representations are more useful, e.g. within requirements elicitation processes. This allows non-developers to create models without learning a completely new language.

Since some parts of the required DSL must allow detailed modelling and the targeted adopters of our approach are developers who are used to structured programming in the domain, a textual language seems to be more suitable. Moreover, as we also address developers with less technical knowledge we propose a new language which incorporates the domain features instead of an API in the programming languages of the domain.

Frameworks like Xtext provide tools for language definitions based on EBNF-based grammars. This allows the development of abstract and textual concrete syntax in one task. Additionally, a text-based editor can be generated based on the grammar definitions. This decreases the required development effort and ensures earlier delivery of the MDE tools. However, if a visual concrete syntax is needed, tools like Eclipse Sirius [230] and EuGENia [229] can be used to augment a textual DSL by a visual editor. These tools allow the development of visual editors based on existing abstract syntax definitions in different formats, e.g. Ecore models (cf. Section 3.3.3).

Acceptance Criteria:

In order to fulfil this requirement, a text-based editor has to be developed. This editor must consider the defined language features of the DSL, to allow developers the specification of text-based definitions of WCMS extension models.

Requirement: User Support Features

Based on the previously described requirement, the text-based editor must ensure a supportive usability. This includes editor features like *syntax highlighting* and *auto completion* to support modellers during modelling actions in order to learn and ensure the required syntax of the language. *Formatters* must get a particular attention during the editor development, since they play an important role for automatically generated extension models, e.g. after a model extraction of existing WCMS extension code. *Validators*, which should include the well-formedness rules of the modelling language, are significantly involved during the validation of existing extension models after a DSL refinement. Therefore, the validator part of the editor should be extensively maintained during the whole infrastructure life cycle.

Fortunately, most of the features can be automatically generated, if frameworks like Xtext and Xtend are used during infrastructure development. If a generated editor feature is not sufficiently implemented, infrastructure developers are able to augment it by additional elements. For example, the auto completion intentionally only contains the elements, which are defined in the DSL's grammar. If additional elements should be provided by the editor, they can be defined within protected regions in form of respective classes for the specific feature. During the code generation of the infrastructure itself, these additional elements become incorporated into the auto completion feature homogeneously. The same applies to other features like validators, which can be augmented by OCL definitions.

Acceptance Criteria:

This requirement is satisfied, if the model editor supports modellers by syntax highlighting, auto completion, and error validation. Additionally, a (set of) formatter(s) must be provided to increase the readability of automatically extracted models.

Requirement: Platform Independence

The previously described frameworks for MDE infrastructure development mainly address modelling in the Eclipse IDE. However, as interviews with extension developers reveal, Eclipse is not the IDE of choice during extension development. The most preferred IDEs are the ones, provided by JetBrains (e.g. IntelliJ IDEA [103] and PhpStorm [104]). Therefore, the minimum requirement is the support of serialization formats like XMI, to transfer models between different IDEs. A much more sophisticated support by explicit plugins for different IDEs is the preferred solution.

Again, using Xtext, facilitates the development of plugins for various IDEs such as Eclipse and IntelliJ IDEA in one step. Additionally, a rudimentary web IDE (or cloud IDE) can be developed. This web editor can provide a foundation for implementing a platform-independent web-based IDE. The advantage of an independent web-based platform can reduce technological bias of extension developers. Developers can directly start modelling and create extensions on the fly within a suitable development setting for the problem domain, since they do not have to deal with the typical installation effort in an IDE. Due to the fact that both approaches, local IDE plugins and web IDEs, have specific developer-centric advantages and disadvantages, solutions for both approaches are required.

Acceptance Criteria:

The acceptance criteria for this requirement includes the successful implementation of a set of plugins for various and common IDEs such as Eclipse, IntelliJ IDEA, and PhpStorm. Additionally, a complete platform-independent IDE should be implemented.

4.3.3 Code Generator

A sophisticated MDE infrastructure for WCMS extensions requires the implementation of one or more code generators. However, especially the development of code generators requires tremendous effort by infrastructure developers. In order to follow an agile bottom-up development approach, as described in Section 3.3.6, infrastructure developers have to inspect elaborated reference extensions. Due to variations in the required extension structure, the development of new reference extensions is inevitable. This main requirement results in the following developer demands dealing with code generators for WCMS extensions as part of an MDE infrastructure.

Requirement: Full Installable Extensions Generation

Usually, WCMS extensions implement a required architecture allowing them to be directly installable to a running WCMS instance. This allows developers to distribute their extensions directly to website integrators, i.e. WCMS administrators who are responsible for the supported features of a WCMS instance. Developers stated, that they expect installable extension packages, if they use an MDE infrastructure. Additionally, extensions must support the representation of the extension data by different representation kinds, e.g. views in Joomla components. As described in Section 4.1, we conducted a study on the Joomla extension directory and investigated that 68% of the views we inspected were standard CRUD views. This amount was confirmed by industrial practitioners who stated to have more than 90% of CRUD views in some of their extensions. Therefore, the main requirement to the generator is support for fully functional and installable extension packages without the need of any handwritten code. This includes the guarantee of interoperability between the generated extension and the core system where it will be installed. So, the required effort to ensure high quality in WCMS extensions (cf. problem statement 1) can be addressed. This requires the implementation of adequate reference extensions which can be used as reference by tool developers during generator development. These reference extensions have to be further developed during the whole life span of the generator in order to follow the defined development process of this work.

Acceptance Criteria:

This requirement is fulfilled, if the generator creates installable extensions which allow the management of specific data (CRUD) in the required extension format of a WCMS. This includes the implementation of adequate reference extensions.

Requirement: Adhere to Coding Styles and Architectural Standards

A requirement to the internal code quality is the generation of extension code, which fulfils a set of coding standards of the respective WCMS community. The generator templates must adhere to the coding standards of every respective programming language of the generated artefacts. For Joomla, a documentation of the required coding standards can be found at [166] but does not fulfil the needs of all extension developers in the domain. In our conducted expert interviews, developers stated that they rather follow official coding standards, such as the PSR-12 coding style recommendation [179]. So, this requirement includes the determination of an adequate coding standard for the programming languages for which code is generated.

Additionally, the generated code architecture must follow the given API pattern of the underlying WCMS. In the case of Joomla, a specific model-view-controller (MVC) pattern on file and code level (see Sect. 2.2.1) must be adhered in order to generate Joomla components. This pattern must be implemented to ensure a proper installation and operability between the WCMS instance and the extension and reduce effort to ensure extension quality (cf. problem statement 1).

Acceptance Criteria:

In order to fulfil this requirement all implemented code generators must ensure the adherence of relevant coding standards within the generated extension code. Additionally, required architectural patterns like MVC must be implemented by the generated extensions.

Requirement: Partial Code Generation

Developers stated, that they are satisfied with their already developed legacy extensions, but would use an MDE infrastructure for further implementation of new extension parts without creating the whole extension anew (cf. scenario 4 in the previous Section 4.2). Therefore, the generator must allow the partial generation of code fragments, which can be directly integrated into an existing extension. Fortunately, the related technologies in the domain of WCMSs allows the conduction of partial code generation. Since developers mainly use interpreter languages, a partial code generation can be easily integrated into the development process of WCMS developers. The generator templates must generate the correct dependencies to existing extension parts in order to ensure a homogeneous functioning of the whole extension after deploying the new code (e.g. by copy&paste) to the legacy extension. This in turn requires the possibility of creating relations of new to legacy artefacts on model level (see requirement *Configurable Generator Hooks* above).

Acceptance Criteria:

This requirement is fulfilled, if the code generator is able to proceed partial code generations. These code fragments must fulfil the requirement to be fully functional to be deployed to existing legacy extensions. If defined on model level, relations between new and existing legacy code artefacts must be generated to ensure proper interoperability between new and legacy code.

Requirement: Support for Smart and Dummy Models

MDE infrastructures often differ in the amount of generated features in dependency to the given model information used during code generation. Some generators require detailed and accurate models to create adequate software fragments (dummy approach). Others can generate most of

the code with abstract model definitions (smart approach). During the interviews with extension developers (cf. Section 4.1 above), the needs for both approaches arose. If developers want to create views like standard CRUD views, a smart generator is required, which can generate the respective code with a minimum of model information. The drawback of a smart generator is the required effort during its development and maintenance, since the whole logic for the implicit code generation must be implemented, mainly resulting in extensive generator templates with a tremendous amount of standard code.

If a more individual solution is required, developers have to create more enhanced models. The generator then must be able to generate the code for these specific needs, but must not create most of the code implicitly. An example for this case can be the need for individual view parameters, which have to be configured within the administration of the extension after it became deployed to a WCMS instance. Whereby a smart generator creates a set of standard parameters, a dummy generator should create the parameters as they are defined in the model.

Both approaches have their advantages and drawbacks, but should be provided by the WCMS-specific MDE infrastructure. Therefore, the generator for WCMS extensions must allow both the implicit generation of standard code fragments based on lean abstract models and the generation of code elements in accordance to enhanced model definitions.

Acceptance Criteria:

This requirement is fulfilled, if the generator is able to use abstract and enhanced models as input during code generation. If an abstract model is given, the generator must create extensive extension code implicitly.

Requirement: Custom Code Support

Developers stated, that they require support for individual functionality. Even though most of their extensions consist of standard CRUD views, individual fragments should be considered during code generation. This includes the realisation of an adequate mechanism to support extension developers during manual code adjustments, i.e. they need to know, which parts of the code can be changed without breaking the whole extension.

As described in Section 3.3.5, different approaches dealing with custom code arose over the last years. The most simple one is the generation of placeholders for custom code (protected regions). During a new generator run, handwritten code within these protected regions will not be overwritten. A better approach such as the 3-tier implementation presented in [221, p. 160 ff] should be considered. The presented approach considers different kinds of relations between generated and handwritten code by implementing architectural patterns providing separation between these both code kinds. Another approach is the use of code models, which deals with individual code on model level. However, maintaining solutions following this approach can exceed the effort of working with the much more simpler approach of protected regions or architectural designs.

Matured frameworks for the development of code generators support tool developers during the implementation of these different approaches. However, finding the right approach for this work has to be determined as part of this requirement, since all come with different advantages and drawbacks.

Acceptance Criteria:

To fulfil this requirement, an adequate approach which supports extension developers during the development of custom features has to be determined and incorporated to the MDE infrastructure. Custom code should be integrated to the generated code in a proper manner without the chance of being lost during further generation runs.

4.3.4 Reverse Engineering Facilities

Some of the collected development scenarios require support for reverse engineering (RE) during the development of WCMS extensions (cf. scenario 2, 3, 4, and 5 in Section 4.2). As developers want to reuse existing legacy extensions or create new extensions that depend on existing third-party extensions, it is necessary to provide a mechanism that raises the extension information to model level.

Requirement: Extract Model Information of Existing Extensions

Some development scenarios are based on the consideration of existing legacy extensions. If an extension evolves, dependencies to existing extension parts such as a data access object or view templates could occur. Understanding existing extensions and managing dependencies to them is a tedious task for extension developers (cf. problem statement 4). To lift this dependency to model level within MDE can only be done, if the required extension information is part of the model. Another case is the development of a new extension, which depends on a third-party extension. A mechanism to lift up the required information of the third-party extension is needed to define the dependencies on model level within MDE.

To support these and other use cases, we propose a semi-automatically extraction of existing extension information in contrast to a completely manual reverse engineering process. Therefore, an adequate mechanism for the extraction of model information of existing legacy extensions has to be implemented. This requires the implementation of model discoverers for the used programming and mark-up languages of WCMS extensions (PHP, HTML, JavaScript, SQL). Matured tools like MoDisco [29, 30] can be used for the extraction of Java-based application information, but lack in support for current web-based technologies as required in the WCMS domain (cf. problem statement 5). Even though discoverers for PHP code exist, the supported language versions have been tremendously outdated. So, a tool for the model-driven reverse engineering (MDRE) for modern (versions of) web-based languages has to be implemented. This tool must at least allow the extraction of the minimal extension information, required for the realisation of common development use cases such as (partial) augmentation (scenario 2 and 4), migration (scenario 3), and modernization (scenario 5) of legacy extensions.

Acceptance Criteria:

To declare this requirement as fulfilled, a mechanism for the model extraction for legacy extensions has to be implemented. The minimal required extension information for the creation of new dependent extensions or extension parts has to be extracted and lifted up to model level into a model instance of a suitable WCMS-specific modelling language.

Requirement: Extract Extensions of Running WCMS Instances

Understanding the code of a deployed legacy extension can become a difficult task. Once deployed to a WCMS instance, corresponding extension fragments may become spread over the whole application, as it is the case for Joomla extensions. Additionally, database entries which bind the installed extension to the rest of the WCMS instance are created. So, in addition to the previously described requirement, developers stated, that they require a mechanism to extract an installed extension from a running WCMS instance. The result of this extraction should be an installable extension package which can be used as input for a further manual or automatic code understanding and model extraction.

Acceptance Criteria:

To fulfil this requirement, a mechanism for the automatic extraction of a deployed extension can be processed on a running WCMS instance. The result of this process should be an installable WCMS extension package for the respective WCMS.

4.3.5 General Infrastructure Requirements

In addition to the previously described requirements, we define overall infrastructure requirements, considering the quality assurance of the infrastructure itself, the adequate integration into the development process of extension developers, and support for tool developers during maintenance of the infrastructure.

Requirement: Quality Assurance

By definition, quality reflects the defined characteristics of a product, service or system. This also applies to MDE infrastructures. However, the problem with quality is its subjective definition by different stakeholders. It can deviate strongly between different persons, because they are measured according to differing personal standards. In [257], Weinberg gives good examples of the different views on software quality. Software users reflect the fast development of an error-less system as good quality, whereas software developers usually define elegant code as indication for high quality. The ISO/IEC 25000 (or SQuaRE²) quality model divides these different views into *internal* and *external software quality* [97]³.

Internal quality considers software quality from the view of a developer, in our case tool developers. Tool developers require well-designed (meta-)models and clearly readable code, which can be simply maintained and extended. Additionally, the testability of the infrastructure must be given to allow developers to ensure correct functionality of their software artefact. Infrastructure artefacts which do not fulfil these requirements may induce unpredictable side effects and system behaviour, each time a change is made. This applies to the DSL, model editors, code generators, and reverse engineering facilities, as well as to the exemplary showcase models. *External quality* describes software quality from the customers view, or in our case the view of extension developers who will use the MDE infrastructure. The main focus of external quality is the quality of software as product and the quality of the building process. If a bug or another defect appears, extension developers may declare this as loss of quality, even though the defect has no effects to other parts of the infrastructure. The same applies to other non-functional requirements such as software usability (e.g. simple DSLs and user-friendly editors) and performance (e.g. dealing with large models during modelling and code generation).

In addition, the internal and external quality of the generated artefacts have to be considered as well. Actually, this is a special requirement for the code generator, which has to ensure generated code with high internal and external quality. Extension developers require generated artefacts with high maintainability and generated tests for the assurance of external quality. Though, we treat and evaluate this requirement as part of the non-functional quality assurance requirement.

Another extending set of quality requirements for MDE infrastructures can be found in [221, p. 366 ff]. The defined requirements, like a well-defined architecture, a stringent programming model, and an up-to-date documentation, will also become incorporated into this requirement. In [55] the authors specify an evaluation method based on existing quality standards and best practices. This method is evaluated by an exemplary sample application. The evaluation method will be considered during further evaluation of the MDE infrastructure, presented in this work.

Acceptance Criteria:

This requirement is fulfilled, if assurance techniques for the internal and external quality of the MDE infrastructure are provided. This includes the assurance of high quality for different quality aspects like maintenance, performance, testability, and usability. Moreover, the internal and external quality of the generated artefacts has to be ensured by the code generators of the infrastructure.

²SQuaRE: Software engineering-Software product **Q**uality **R**equirements and **E**valuation.

³This straight definition has been combined in 2011 to product quality, but both aspects are still considered separately during quality measurement within the latest ISO/IEC 25000 standard version.

Requirement: Integration into existing Extension Development Processes

In addition to the previously defined requirement, another overall requirement is the integration of the MDE infrastructure into the development process of extension developers. To this end, the MDE infrastructure should be usable in combination with other MDE tools for documentation, versioning, and testing. So, developers can use it within different stages of extension development. By providing IDE plug-ins which include a model editor and transformation tools, developers can incorporate the infrastructure to their common development environment. Moreover, strategies should be elaborated for a proper interaction of the MDE infrastructure with a test suite and a continuous integration infrastructure.

The exchange of the project artefacts between the MDE infrastructure and other tools can be achieved through serialisation mechanisms, e.g. for model instances. Due to varying development procedures, the infrastructure must allow a certain variability. Some developers stated that they exclusively work within their preferred IDE, whereas other developers integrate various tools and IDEs within their common development process. Therefore, the MDE infrastructure for WCMS extensions must be flexible enough to be suitable for different development processes.

Acceptance Criteria:

To fulfil this requirement, the infrastructure must be flexible enough to be integrated into various development processes of extension developers. In addition, sample scenarios should be applied and documented in order to simplify process integrations and allow portability to industry.

Requirement: Automatic Build of MDE tools for Different IDEs

Another overall requirement is the support of tool developers by automatic build mechanisms for the infrastructure artefacts. The support of different IDEs and the provision of a web editor requires unnecessary effort during repetitive plug-in packaging. This effort can be saved by providing automation mechanisms, e.g. as part of a continuous delivery pipeline including automatic test execution and plug-in packaging for all supported IDEs. Additionally, continuous deployment of web-based cloud IDEs should be realised to support infrastructure developers.

Acceptance Criteria:

In order to fulfil this requirement, automation mechanisms must be provided to support tool developers during the creation of various IDE plug-ins and the web editor.

4.3.6 Summary

Table 4.7 summarizes the elicited requirements. In order to address the requirements in this work, unique identifiers based on the subject are added. Moreover, we added the addressed problem statements which were defined in Chapter 1 to the requirements. Some requirements necessitate tool implementations in order to avoid missing tool support for our approach. Therefore, we refer such occurring requirements to problem statement 5 (parenthetic), even though they do not directly address the problem of missing supportive tools for WCMS extension development.

4.4 Discussion

In this section, we discuss how adequate RQ1 (*How can MDE support common WCMS Extension Development Scenarios?*) and its related three sub questions were answered in this chapter. RQ1.1 concerns the relevance of the previously presented development scenarios (Section 2.3), whereas RQ1.2 is targeted on an MDE concept for these scenarios. The follow-up question RQ1.3 aims at MDE infrastructure requirements for adopting MDE during WCMS extension development.

Table 4.7: MDE Infrastructure for WCMS Extensions - Requirements

Subject	ID	Requirement	Problem Statement
Domain-Specific Language	R1.1	Data Modelling	1, 2, 3, 4
	R1.2	Data Representation and Interaction Modelling	1, 2, 3, 4
	R1.3	Extension Modelling	1, 2, 3, 4
	R1.4	WCMS Core Support	1, 2, 3
	R1.5	Configurable Generator Hooks	3, 4
	R1.6	Quality Assurance of Extension Models	(5)
Model Editors	R2.1	Textual Model Representation	1
	R2.2	User Support Features	(5)
	R2.3	Platform Independence	5
Code Generator	R3.1	Full Installable Extension Generation	1, 2, 3, 5
	R3.2	Adhere to Coding Styles and Architectural Standards	1, 2, 5
	R3.3	Partial Code Generation	3, 4, 5
	R3.4	Support for Smart and Dummy Models	1
	R3.5	Custom Code Support	1
RE Facilities	R4.1	Extract Model Information of Existing Extensions	3, 4, 5
	R4.2	Extract Extensions of Running WCMS Instances	5
General Infrastructure Requirements	R5.1	Quality Assurance	1, 2, (5)
	R5.2	Integration into existing Extension Development Process	5
	R5.3	Automatic Build of MDD Tools for Different IDEs	(5)

4.4.1 Relevance of Scenarios 1-5 (RQ1.1)

In order to ensure the relevance of the elicited development scenarios (RQ 1.1), we conducted a set of semi-structured expert interviews with industrial practitioners from the WCMS domain. Based on the interview results, *the relevance of our predefined scenarios is strongly confirmed by the statements of the interviewed industrial practitioners* (see Section 4.1). Though, despite the promising results, they are subject to a number of threats to validity (cf. [260]).

While the expert interviews confirm the crucial role of the elicited scenarios, the participants also pointed us to an additional scenario that we did not consider yet. In particular, the abstraction of shared functionality into libraries. This is a threat to *construct validity*, since we did not consider this specific case during the preliminary scenario elicitation in Section 2.3. We aim to study this case in future work. Another threat to construct validity is due to our interview guide. Even though the interview results confirm that iteratively refining already existing extensions (extension evolution) is a challenge, explicit questions considering this related sub-scenario are not part of our interview guide. Further interview iterations should incorporate such questions in order to get the interviewees opinion of this sub-scenario which is included in scenario 1 (development of independent extensions) and scenario 2 (development of dependent extensions).

An *internal validity* threat is given by the study design and the set of interview questions. The initial study design was rather qualitatively in order to gain insight of the WCMS extension development procedure of industrial practitioners in an inductive way. Therefore, the questions were not directed (directly) to the development scenarios.

The interview results were interpreted in order to confirm the relevance of the scenarios and their potential for MDE. However, the relatively small sample size of 8 participants leads to a potential threat to *conclusion validity*.

A threat to *external validity* is based on the involved interviewees. All experts are developers for the Joomla WCMS with little or no experience in extension development for other WCMSs. It has to be studied if the scenarios are also common during development of extensions for other WCMSs. However, the conducted interviews fit into the scope of this work, since we consider Joomla as reference WCMS (cf. Section 2.2.2).

4.4.2 MDE Concept for Scenarios 1-5 (RQ1.2)

Based on the elicited scenarios 1-5, we defined concepts for the adoption of MDE as alternative to conventional WCMS extension development. These concepts show, how MDE can support developers during scenarios 1-5 as answer to RQ 1.2. To this end, interactions between the involved roles (extension developer, administrator, content manager) and supportive MDE infrastructure facilities were incorporated into MDE process definitions. The presented concepts consider the common MDE infrastructure facilities for forward and reverse engineering tasks. These are in accordance with used infrastructures of already adopted MDE approaches (cf. Section 3.2) and are in compliance with the definitions of established literature such as [221] and [28].

As the interview results offer, the elicited scenarios are relevant and offer high potential for an MDE approach. Therefore, they are providing a good basis for MDE concepts. For the sake of completeness, additional scenarios should be elicited in future work. So, construct validity of further empirical research based on these concepts can be ensured. Though, the presented concepts should be adaptable for additional scenarios, based on the referenced forward and reverse engineering facilities, as well as the given user roles of the domain. In order to ensure external validity, we kept the concepts abstract without dependencies to a specific WCMS.

4.4.3 MDE Infrastructure Requirements for Scenarios 1-5 (RQ1.3)

To adopt the defined MDE concepts from Section 4.2, we collected the requirements for a suitable MDE infrastructure as answer to RQ 1.3 (Which facilities are required to achieve MDE during these scenarios?). These requirements consider all necessary facilities including the demands of extension developers based on the expert interviews. With regard to the research of MDE in practice during WCMS extension development, the requirements include acceptance criteria for the evaluation of a suitable MDE infrastructure. However, the requirements elicitation of this chapter is rather abstract in order to get the overall requirements for the MDE infrastructure. Typically, requirements engineering consists of several explicit processes such as described in [136] and [175]: elicitation, analysis and negotiation, documentation, validation, and management. So, all necessary requirements can be collected at the beginning of a software project. Though, in the scope of this work, abstract requirements are sufficient for MDE infrastructure development, since we follow the agile bottom-up development process as proposed in [244] (cf. Section 3.3.6). More specific requirements occur during the development process and extend the requirements list in an agile manner. Moreover, the general requirement definition is sufficient for the discussion of existing MDE infrastructures for extension development.

In [26], Boehm presents guidelines for the verification and validation of software requirements. Hereafter, we discuss the presented requirements based on these guidelines in order to ensure a sufficient requirements coverage. To be more specific, we assess their completeness, consistency, and feasibility (cf. [26]). Based on the description of Boehm, all parts of a system must be considered and described by requirements to ensure *completeness*. Focusing on the presented MDE concepts, all requisite infrastructure components are covered by our requirements definition. This includes the main facilities for modelling (DSL, model editor), forward engineering (generator templates), and reverse engineering (model and extension extractor). All requirements consist of acceptance criteria and do not contain placeholders or non-existent references (cf. [26]). In order to achieve *consistency* within the requirements specification, conflicts must be avoided. This

aspect is covered due to the limited set of requirements, but must be considered during further requirement amendments. With the *feasibility* aspect, Boehm covers non-functional requirements and the identification of high-risk issues considering the life-cycle of a system. This includes maintainability, reliability, testability, and portability. These aspects are covered by our overall MDE infrastructure requirements, since we consider the integration into existing development processes by suitable IDE plugins and model serialisation mechanisms, as well as testing and rapid delivery of the infrastructure by continuous integration/delivery/deployment support. In addition, we suggest the use of popular meta frameworks for the realisation of the infrastructure itself. So, the costs during development and maintenance can be minimized. A detailed overview of common risks which are related to cost-effectiveness, can be found in [26].

5

Domain-Specific Language for WCMS Extensions

*Any program
is a model of a model
within a theory of a model
of an abstraction of some portion
of the world or of some universe of discourse.*

– Meir "Manny" Lehman

In this chapter, we present a domain-specific modelling language for WCMS extensions. After discussing related work and existing suitable approaches, we demonstrate the language features based on the elicited requirements as defined in the previous chapter. The presented language is the cornerstone of our proposed MDE approach for WCMS extension development. Therefore, the language is the main requirement in order to address the problem statements as described in Chapter 1. This also includes the required tool support for the modelling process (cf. problem statement 5). Therefore, well-formedness rules for the language, corresponding model editors, and showcase models are introduced. Finally, we will evaluate the DSL and related editors according to the requirements presented.

5.1 State of the Art

Adopting MDE in the web domain is not a completely new approach. Languages for general web development like *UWE* (UML-based Web Engineering) [137], *WSL* (Web Specific Language) [225], *WebML* (Web Modeling Language) [42, 41], *WebDSL* [247], and the *IFML* (Interaction Flow Modeling Language) [27] can be used to create models of complete functionally-rich web applications. It is worth mentioning, that the IFML language is standardized by the OMG (cf. [164]) and is popularly used for web and mobile application development, e.g. with WebRatio [256]. Though, these languages are not suitable for the elicited and confirmed development scenarios which are stressed in this work, since they do not address WCMS extension development (UWE, WebML, WebDSL, IFML) or are intended for other systems like Zope and Plone (WSL). However, the basic language features for data and interaction modelling can be adapted to WCMS-specific DSL requirements as presented in Section 4.3. Therefore, we will incorporate those ideas that might suit to the WCMS domain as well.

In addition to existing DSLs for general web applications, only a few approaches for MDE adoptions in the WCMS domain, considering its specific objects and features, exist. In [248], Vlaanderen et al. research the adoption of the OOWS method [67] in the WCMS domain using the *OOWS Method Metamodel* as DSL. The authors conclude, that the expressiveness of the meta-model is not sufficient for WCMSs. Trias proposes a meta-model (*CMS Common Metamodel*) for WCMS platforms based on an analysis of popular WCMSs in [235]. This meta-model concludes basic WCMS features which are content, user, navigation, and behaviour. Saraiva et al. propose the *CMS-ML* language for concrete WCMS instances [204, 205]. This meta-model includes the common WCMS platform features and allows the creation of WCMS instance templates. A drawback of the language, as discussed by the authors, is its missing adequacy for content-centric WCMSs like WordPress, Joomla, and Drupal.

The ReLiS framework, presented by Bigendako et al. in [25], is considered as a specialized CMS with extension capabilities. It supports researchers during collaborative conduction of systematic review (SR) projects. The authors present a DSL, a web-based editor, and further tool support to automatically build, install, and (re-)configure individual SR projects as extensions to the ReLiS platform. The authors propose an automatic deployment of generated extensions, which is a promising support for our proposed infrastructure as well. However, the presented approach operates on actual extension instances in a running application similar to the previously described works, whereas our approach addresses extension development on a higher abstraction level.

Apart from these works, there is little recent research on the development practices for WCMSs, an observation that is confirmed by Norrie et al. [160]. Most of the presented approaches propose platform-independent meta-models for the development of specific WCMS instances. Though, none of these works addresses the extensibility of WCMSs through standardized extension types taking their interdependencies into account. Furthermore, these approaches focus on support for WCMS administrators, since they only consider use cases of WCMS-based web applications. MDE support for extension developers during development of additional functional features is not in the scope of these approaches. Additionally, dependencies between newly developed and existing extensions are not provided in any of these works. Though, the works are relevant for the interaction of functional extensions and existing WCMS core features (cf. Section 5.2.4).

Only the work of Filipe et al. represents an adequate and up-to-date approach to close the gap of missing MDE approaches for WCMS extensions. In [66] the authors present the *XIS-CMS* language, a platform-independent DSL for WCMS extensions. The DSL is based on a UML profile which supports WCMS developers by a smart and dummy approach. More specific, developers can choose between small models whereas most of the application is generated automatically (smart) or enhanced and detailed models which are used to create more individual extensions (dummy) [193]. Even though the DSL is intended as platform-independent solution, it is strongly evaluated for the DotNetNuke WCMS, which keeps less than 0.2% market share. Even though this WCMS is not that widely used, it has gained popularity in recent years, since it is based on Microsoft's ASP.NET platform. Another drawback of the approach is the missing evaluation. As this WCMS has a limited extension mechanism like WordPress, another language, providing suitable abstractions and automation facilities for a more sophisticated extension mechanism is needed. So, we further research the usefulness and profitability of MDE in the WCMS domain concerning a more popular WCMS with a more sophisticated extension mechanism.

Addressing the challenges during WCMS extension development and researching the appropriateness of MDE requires a suitable DSL for the high level description of WCMS extensions. However, none of the previously described works proposes a suitable DSL which can be used during MDE of the elicited development scenarios 1-5. Existing DSLs do not provide suitable abstractions for a more sophisticated extension mechanism as the one implemented in the Joomla WCMS. Moreover, they do not consider dependencies between extensions. Therefore, we propose a DSL for WCMS extensions hereinafter, taking these aspects into account. In addition, we provide tool support for this DSL in form of model editors which can be integrated in the development process of developers in the domain.

5.2 Language Design

In accordance to France et al. [72, p. 43f], the two main challenges during DSL development are the *abstraction challenge* (How can one provide support for creating and manipulation problem-level abstractions as first-class modelling elements in a language?) and the *formality challenge* (What aspects of a modelling language's semantics need to be formalised in order to support formal manipulation, and how should the aspects formalised?).

During the last decade two different, but not excluding, approaches, addressing these challenges, have been established. The first approach is to implement a completely new language with respect to the domain which has to be modelled (external DSL). Another approach is to create a modelling language based on a general purpose language (embedded or internal DSL) [71]. This allows a more rapid development of the language and decreases the implementation effort, since most parts of the host language are reused. In order to elicit an appropriate kind of DSL (internal or external) and modelling format (text-based or visual) for WCMS extensions, we refer to the requirements of industrial practitioners. This is a requirement for the development of appropriate model editors which shall be used intentionally by extension developers. Since developers require a solution close to a programming language (cf. Section 4.1), a textual representation seems to be the best solution for our approach. Using the Xtext framework allows a straightforward DSL definition for WCMS extensions, based on a grammar-based DSL. This allows to create both, the abstract and concrete syntax in one artefact. So, text-based editors can be created with minimum effort and the addition of visual editors is possible on request.

Based on the extension features of popular WCMSs (cf. Section 2.2.2), we propose the **eJSL** DSL. This DSL consists of three main modelling parts: a part to model the data management of WCMS extensions (**entities**), a part for the definition of interactions between data representations (**pages**), and a part for the structural description of an extension and its metadata (**extensions**).

We decided to define the *entities* and *pages* parts as platform-independent as possible. So, we allow a further use of the DSL, e.g. for other WCMSs. The design of these parts is influenced by the Simple Web Application Language (SWAL) [28], WebDSL [247], and WebML [208]. This language describes the data and interaction of a general web application. The *extension* part is used for the specification of particular WCMS extensions. Since we decided to use Joomla as reference WCMS, based on its sophisticated extension mechanism and support for various extension types, the extension part of the DSL fits perfectly to the Joomla WCMS. However, it can be used platform-independently for the extension description of any WCMS. An overview of the main DSL features can be found in Figure 5.1.

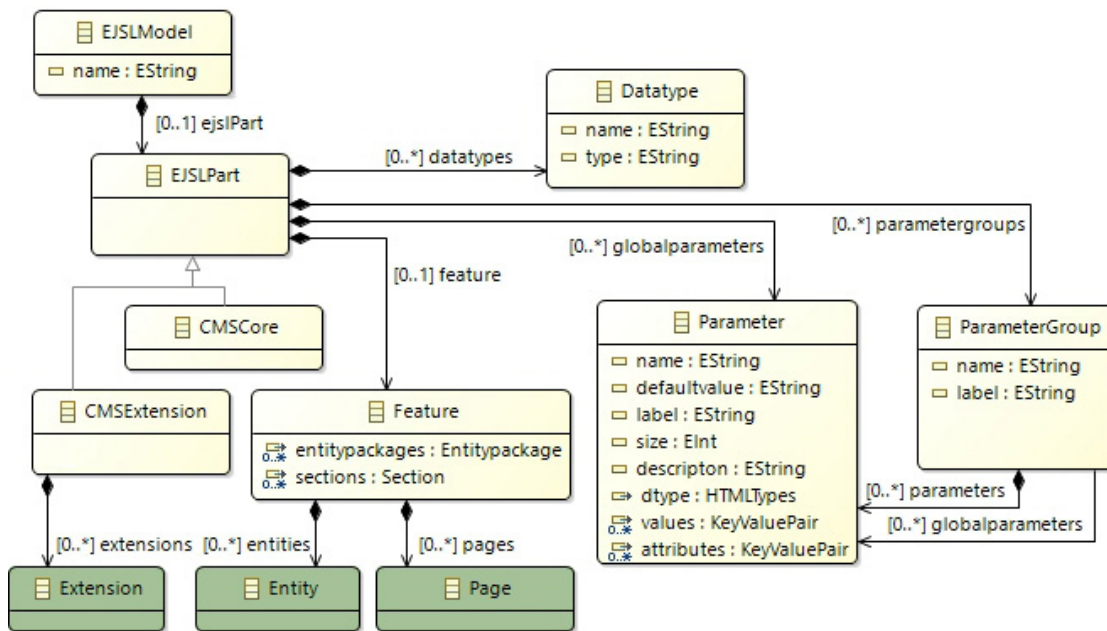


Figure 5.1: Overview of the eJSL Meta-Model

As Figure 5.1 illustrates, the language additionally allows the specification of core elements to create mappings between new extensions and the existing core features such as content, users, and menus. Moreover, language features for the specification of own data types (**Datatypes**) and parameters (**Parameters**) exist. These can be used within entity and page definitions. The latter is used for various parameter description in different contexts. A Joomla component, e.g., can contain specific global parameters for its configuration (mostly called *options*), whereas its contained elements like views can have their own parameters. Figure 5.2 illustrates an example for the options of the users component in a Joomla 4 instance. The eJSL grammar is publicly available as part of the JooMDD project on GitHub: <https://github.com/thm-mni-ii/JooMDD/blob/master/de.thm.icampus.joomdd.ejsl.parent/de.thm.icampus.joomdd.ejsl/src/de/thm/icampus/joomdd/ejsl/EJSL.xtext>.

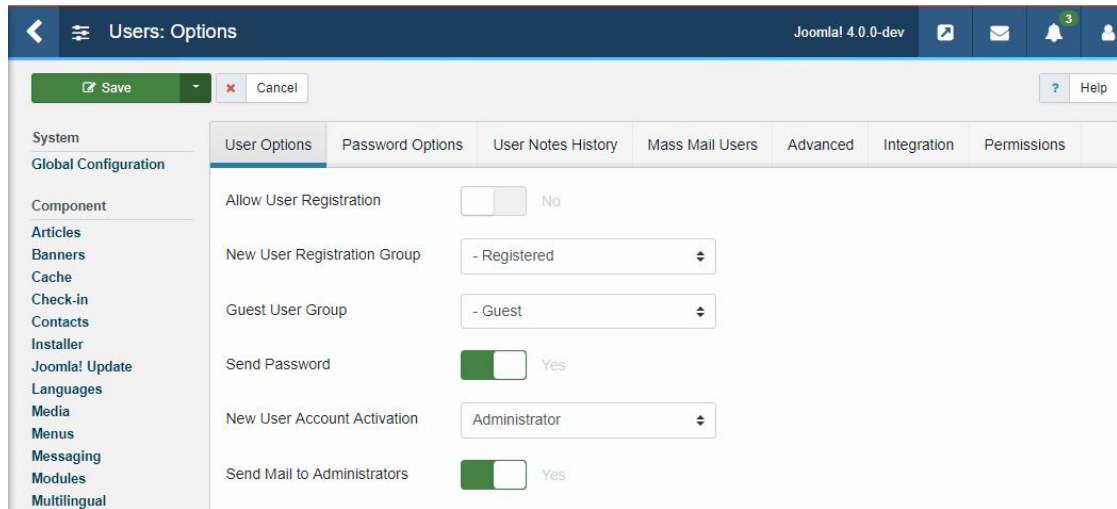


Figure 5.2: Parameters of a Joomla 4 component (Users Component)

Henceforth, we present the language features in detail by illustrating straightforward model examples. For a better understanding we provide instance models and their counterparts as installed extension for each part of the language. *As running example, we use a conference management extension* which is also available as showcase model of the eJSL DSL (cf. Section 5.5). This model represents a conference management component for WCMS-based websites. Once installed, it provides functions to allow the management of common conference entities such as participants, talks, rooms, and programmes. To this end, the component requires management views to allow users with specific rights to perform CRUD actions on the respective entities. Additionally, links between different views must exist to ensure a user-friendly interaction between related component views. Figure 5.3 illustrates an exemplary interaction between the list and details view of the participants view in the backend of the component (running within a Joomla instance). Whereas the list view represents a list of all existing participants, the details view allows to edit a participant. If a user with adequate rights clicks on a participant in the list view, the respective participant must be selected in the related details view. The component provides several filters (table options) for the list views of the respective entity. Details views allow the creation of references between different entities. I.e. a participant can be referred to a talk to be specified as a speaker. To ensure a homogeneous installation, as well as an adequate look and feel, the component must contain meta information like supported languages, authors, and the allocation of the component views to the frontend and backend section. This information is also provided by the conference model. The frontend part of the extension illustrates the existing conference data such as the programme, rooms, and participants (see Figure 5.4). Like the backend list views, the frontend views may provide filters for the illustrated entities.

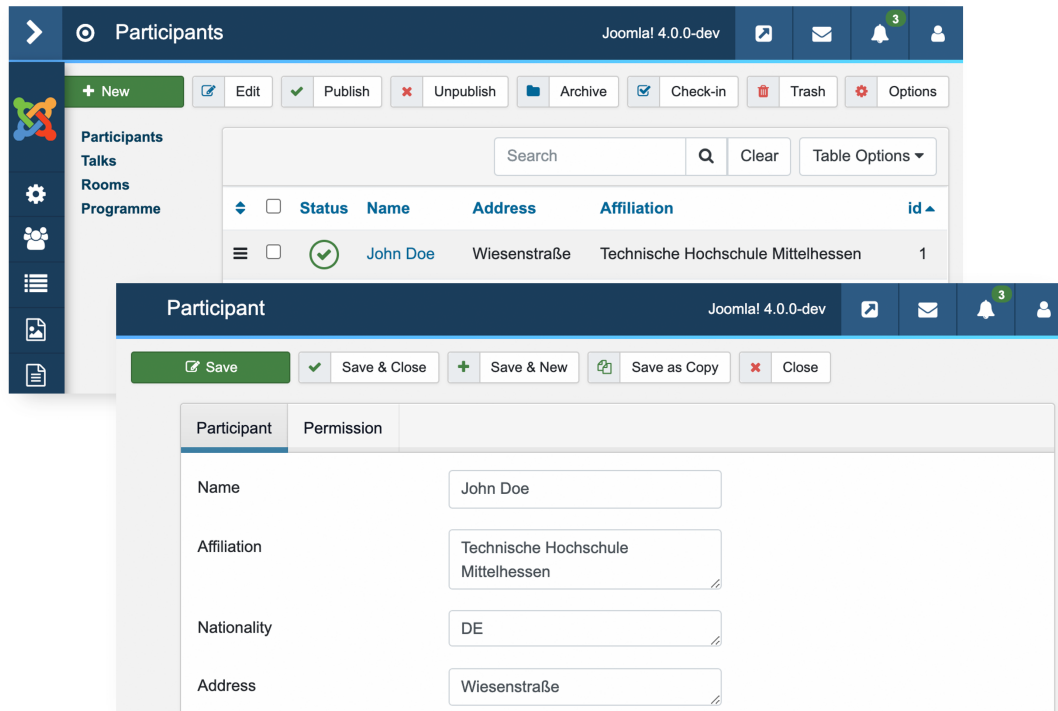


Figure 5.3: List and Details View of an installed Conference Component

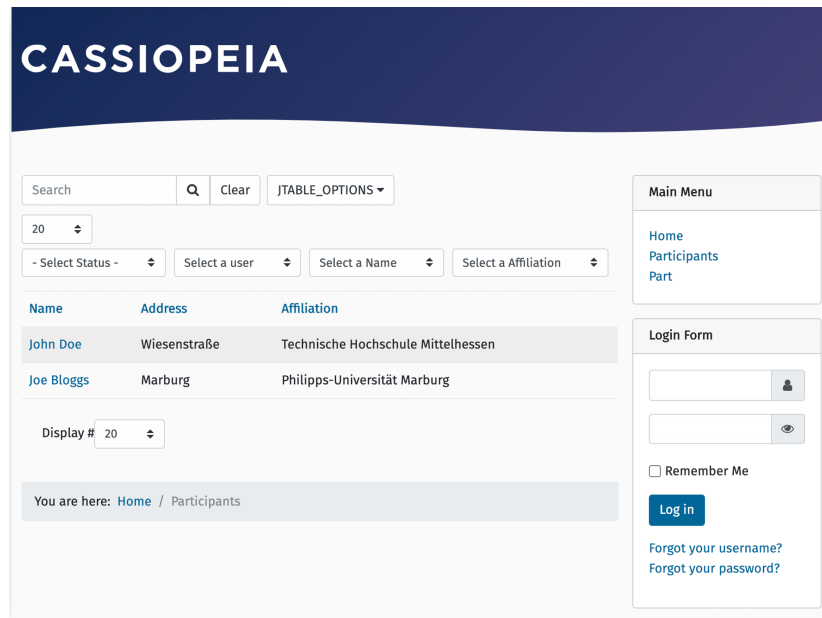


Figure 5.4: List View of an installed Conference Component (Frontend)

Even though we use Joomla as reference within the examples, the concepts of the language are not restricted to this WCMS, i.e. the DSL can be used for WCMS extensions in general. The example conference model is further explained in Section 5.5.

5.2.1 Data Modelling

The data modelling part of eJSL allows developers to define data entities of an extension and their relations to each other. This includes popular language features like *inheritance*, *attributes* and *references*, as well as a *flexible type system*. Additionally, we introduce a *language annotations feature* for model features which can be interpreted by generators during translations to code. Figure 5.5 illustrates the *entities* part of the eJSL meta-model.

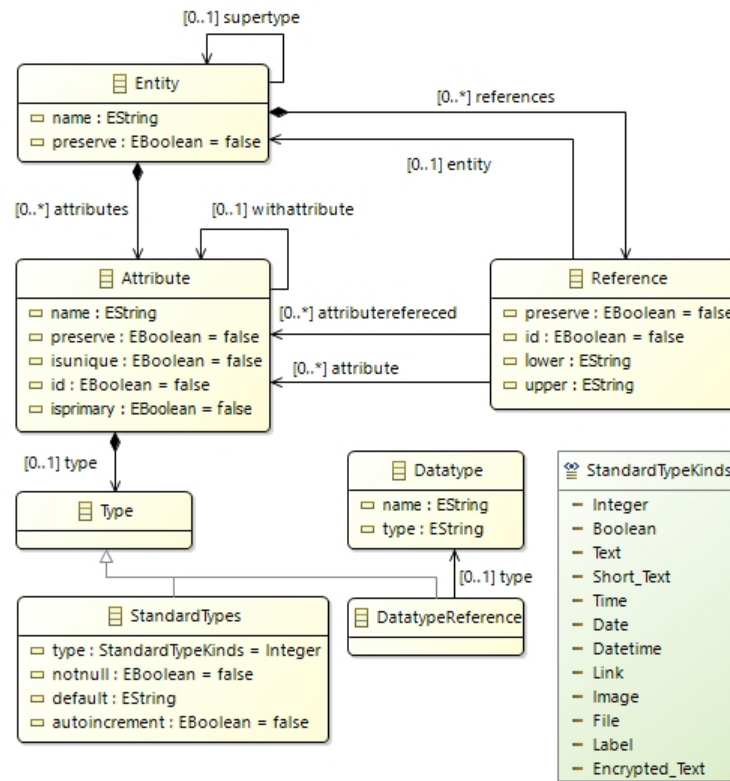


Figure 5.5: Data Modelling Part of eJSL

An entity can contain typed attributes and references to other entities. More specific, a reference represents an attribute-to-attribute relation between two entities. To ensure valid references between existing entity attributes, adequate constraints are implemented (cf. Section 5.3). To allow inheritance, entities can extend each other. Entity attributes can be typed by predefined standard types like **Integer**, **Boolean**, or **Text**. Additionally, modellers can define their own data types which can be used for attribute typing. We decided to choose abstract standard types, since they can become translated to different type representations in generated code. The most straightforward example is represented by the type **Short_Text** for attributes like names or titles. Whereas the type can be translated to **varchar(255)** in an SQL installation file, the representation of the same attribute within a component view may be translated to a HTML text input type (`<input type="text">`). This exemplifies the influence of the entities part during further translations to code. Besides data management of an extension, entities may affect further representation and interaction definitions of the data as well (e.g. CRUD operations).

Listing 5.1 shows an example for inheritance within an entities definition as part of an eJSL instance model. Entity elements can be defined once and passed to extending entities. In the example, the name and address attributes are defined for an abstract **Person** entity, whereas the **Participant** entity extends this entity. During further code translations, the attributes can

be inherited to participants, i.e. code for managing names and addresses of participants can be generated. To avoid a code generation for the abstract entity, we introduce the **@preserve** annotation. This annotation can be used for annotations of various model elements to avoid code generation for annotated model elements. So, the language can support various development scenarios such as scenario 2 (development of dependent extensions) and scenario 4 (partial augmentation of legacy extensions). As the example illustrates, entity attributes can be declared as **unique attribute**. Additionally, attributes can be tagged as **primary attribute**, which may be useful for explicit database-specific definitions. However, in order to avoid the definition of IDs within an eJSL model, unique and primary definitions are optional language features. To prevent the further generation of several unique elements, modellers have full control on how unique attributes are handled in the generated artefacts. The most common use case is the use of unique IDs, which can be generated automatically without an explicit model definition.

```

1 Entity Person @preserve {
2   attributes {
3     Attribute name {
4       type = Short_Text
5       Unique attribute with ID
6     }
7     Attribute address {
8       type = Text
9   }}}
10
11 Entity Participant extends Person {
12   attributes {
13     Attribute affiliation {
14       type = Text
15     }
16     Attribute nationality {
17       type = Text
18   }}}

```

Listing 5.1: Data Model for Conference Extension (Inheritance and Attributes)

In Listing 5.2, an example for the use of references within an entity definition can be found.

```

1 Entity Talk {
2   attributes {
3     Attribute title {
4       type = Short_Text
5       Unique attribute
6     }
7     Attribute speaker {
8       type = Short_Text
9   }}
10   references {
11     Reference {
12       EntityAttribute = speaker
13       ReferencedEntity = Participant
14       ReferencedEntityAttribute = Participant.name
15       min = 1
16       max = 1
17   }}

```

Listing 5.2: Data Model for Conference Extension (References)

An entity attribute can refer to an attribute of another entity. This information can be relevant for further code translation steps (e.g. UI code generation). A generated extension should allow to create relations between existing data entries of different entities. Additionally, the language allows the definition of reference multiplicities. This may affect further representation and validation behaviour within generated extensions. Figure 5.6 illustrates how a reference to another entity attribute can be represented in a generated extension view (installed to a Joomla-based application). During the creation of a new talk, a speaker from a list of all existing participants can be chosen. As the minimum and maximum number of speakers is defined to one, a selection has to be made. Otherwise, a validation error will be shown.

Figure 5.6: Representation of an Entity Reference to another Entity Attribute

5.2.2 Interaction Modelling

To specify the interaction behaviour of a modelled extension, the *pages* part of the eJSL DSL is defined (cf. Figure 5.7). This language part can be used for the definition of various *entity representation kinds* as well as interactions between them by *flexible link definitions* and *page actions*. This part of the DSL adopts the hypertext modelling concepts from the SWAL [28] and WebML [208] languages. A page embodies a representation artefact within a WCMS extension, i.e. a view in a Joomla component or a data representation in a Joomla or Drupal module. The eJSL language provides several *page kinds* to facilitate various representation definitions. A *StaticPage* describes a static HTML representation whereas a *DynamicPage* is the abstraction of the CRUD views of an extension. The latter can be used to define views which represent a list of existing entities (*IndexPage*) and a detailed and optionally editable overview of a selected entity (*DetailsPage*). To allow inferences by code generators, a dynamic page must contain a reference to a defined entity. The fields of a *DetailsPage* can be specifically configured to restrict the (editable) attributes to a subset of all entity attributes which shall be illustrated within a details view. Figure 5.3 above shows a list and details view for conference participants in a Joomla 4 component.

In addition to the previously described dynamic page kinds, the (*CustomPage*) kind is provided, to allow the model description of custom entity representations which differentiate from standard CRUD representations. However, to define the representation form of a custom page, modellers can specify the page kind, i.e. either a list or detail representation, similar to the two *DynamicPage* representations, or a complete custom definition. In the latter case all required files could be created during further code generation but require manual refinements in the generated code.



Listing 5.3 shows an example for a static page representation as part of the conference extension model. Within a `HTMLBody` section the HTML body can be defined, which is used in a corresponding view. This HTML body can contain a complete individual HTML description. In Listing 5.4 an example definition for a custom page is presented. Besides the referenced entity, the eJSL language allows to define the page type. This allows to generate different variants of custom representations. The listing also contains a page action for the action type `SAVE`. Consequentially, adequate code for storing data for the referenced entity can be generated. The (optional) position feature stores the information on how the page action shall be represented in a generated extension. In this example, a button has to be placed on the top of the view.

```

1 StaticPage Welcome {
2   HTMLBody {
3     "HTMLBody"
4   }
5 }

```

Listing 5.3: Page Model for Conference Extension (Static Page)

```

1 CustomPage Speakers {
2   Page type: custom
3   *Entities Participant
4   pageactions {
5     PageAction save {
6       type = SAVE
7       position = top
8     }
9   }
10 }

```

Listing 5.4: Page Model for Conference Extension (Custom Page)

A more sophisticated model extract can be found in Listing 5.5 which shows the definition of a dynamic list representation (IndexPath). The excerpt illustrates a model specification for a list representation of the participant entity. This representation can be interpreted and generated e.g. as list view in a Joomla component which illustrates the attributes of existing entities with context links to a detail view for each list entry (cf. Figure 5.3). As the model excerpt shows, the language allows to limit the table columns explicitly (**representation columns**).

```

1 IndexPath Participants {
2   *Entities Participant
3   representation columns = Participant.name, Participant.address,
4     Participant.affiliation
5   filters = Participant.name, Participant.affiliation
6   links {
7     InternalcontextLink Details {
8       target = Participant
9       linked attribute = Participant.name
10      linkparameters {
11        Parameter name = *Attribute "Participant.name"
12      }
13    }
14  }
15 }

```

Listing 5.5: Page Model for Conference Extension (Index Page)

In addition, modellers can specify the list **filters**. This information can be used during code translation to create filter functionality for the respective view (see Figure 5.8).

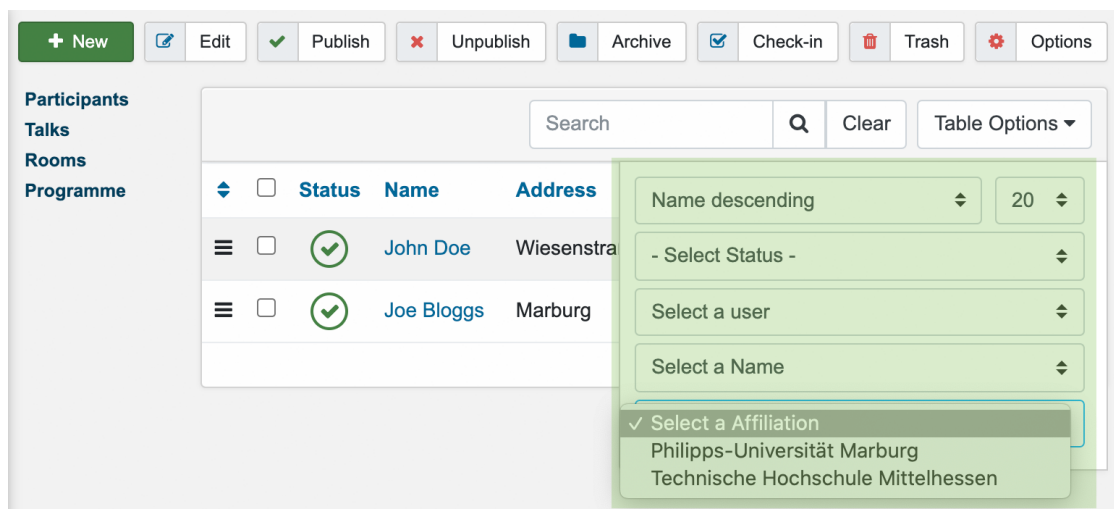


Figure 5.8: Filters for Existing Participants (J4 Backend)

The defined context link shows an interaction example between the specified list of an entity type (Participant) and a detailed representation of a specific entity. The details page definition can be found in Listing 5.6. The context link is specified to be set for the name attribute of each list item with the participant details page as target. Additionally, a parameter containing the name of the selected participant is specified. So, e.g. the name of the clicked list entry will be sent as GET or POST parameter, dependent on the interpretation by a code generator.

```

1 DetailsPage Participant {
2   *Entities Participant
3   links {
4     InternalLink Index {
5       target = Participants
6       linked attribute = name
7     }
  }
}

```

Listing 5.6: Page Model for Conference Extension (Details Page)

The specified details representation for a specific participant includes an internal link back to the participants list page with the identifier **Participant**. This information should be sufficient for a further translation to interacting views providing CRUD of participants.

5.2.3 Extension Modelling

The third part of the eJSL language provides language features for the definition of extension-specific information. So, abstract meta information about the extension, the extension type, referred pages, and the supported languages can be specified (see Figure 5.9).

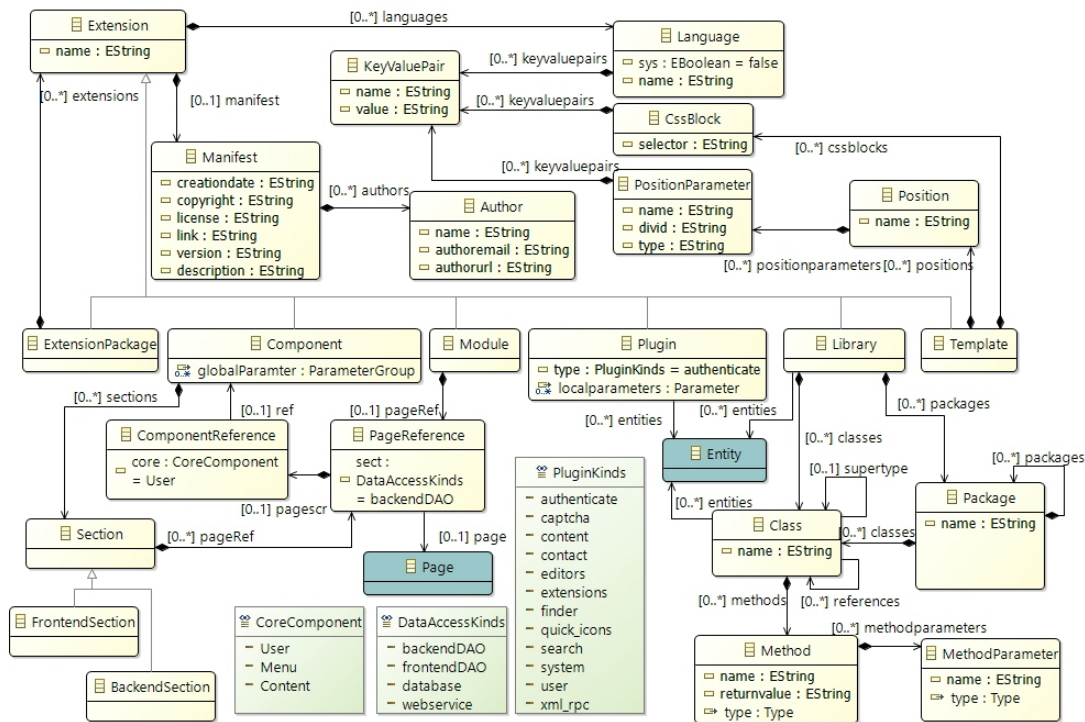


Figure 5.9: Extension Modelling Part of eJSL

As described in Section 2.2.1, the extension mechanism of the Joomla WCMS supports various extension kinds varying in their functionality and intended purpose. This is the major difference and advantage in comparison to other WCMSs. Therefore, the extension part of the eJSL DSL follows the extension kinds supported by Joomla. This allows complex extension definitions, as required for Joomla, as well as specifications for other systems with less complex extension kinds. In order to allow further DSL refinements, we decided to implement a composite pattern for the extensions part. This allows us to easily implement a solution for extension packages which in turn may contain other extension packages, as well as extensions of various extension kinds.

As Figure 5.9 illustrates, every extension definition may consist of a manifest and language specifications. The latter can be used to create key value pairs in different languages for language constants. This feature addresses the internationalisation and localisation feature of WCMSs. Supported languages can be listed in the format `<country code>-<language>`. Some WCMSs support language management in a separated way for the frontend and backend of an instance. To support such a feature, the eJSL language includes a `system` tag for language specifications. The manifest part follows a straightforward approach. It consists of meta information such as author, copyright, licence, and version of the extension. All required fields can be specified or should be filled by default values during further code generations. Listing 5.7 illustrates an example manifest specification, whereas Listing 5.8 shows an exemplary specification of supported languages for an extension.

```

1 Manifest {
2   authors {
3     Author "John_Doe" {
4       authoremail = "john.doe@example.org"
5     }
6   }
7   copyright = "Copyright_(C)_2020_All_right_reserved."
8   license = "GNU_General_Public_License_version_2_or_later;_see_LICENSE.
9             txt"
10  version = "1.0.1"
11 }
```

Listing 5.7: Extension Model for Conference Extension (Manifest)

```

1 languages {
2   Language en-GB
3   {
4     keyValuePairs
5     {
6       SUCCESS_MESSAGE = "This_was_successful!"
7       ...
8     }
9   }
10  Language system en-GB {}
11  Language de-DE {}
12  Language system de-DE {}
13 }
```

Listing 5.8: Extension Model for Conference Extension (Languages)

The variety of extension kinds leads to sophisticated modelling possibilities. Depending on the specified extension kind, extensions can be mapped to existing pages and entities. In particular, Joomla components and modules are extension kinds which mainly consist of views for the representation of any kind of data. Therefore, the language allows optional references between

these extension types and pages. By adding additional information to such a page reference, it is possible to describe dependencies between several extensions in an abstract manner. The **Component**, **Module**, and **Plugin** parts of the language require few mandatory elements as basis for installable software extensions with a large code base, whereas the **Library** and **Template** parts require elaborated specifications for a further code generation.

Components and modules are mainly used for the management and representation of any data. Therefore, the respective extension parts of the DSL are tailored to these features, providing abstract model elements for them. A **Component** specification can contain references to existing pages (see Section 5.2.2 above). These page references can be specified for the respective section of Joomla components (frontend or backend). So, interaction models (**pages**) can be reused within different components and sections of one component. In Listing 5.9 an exemplary page mapping within a component specification is illustrated. For each section of a component (frontend, backend), the existing **participants** pages are referred. This is an example for the reuse of the same page within one component.

```

1 Component MyConference {
2   Manifest {...}
3   languages {...}
4   sections {
5     Frontend section {
6       *Pages {
7         *Page : Participants
8         *Page : Participant
9         ...
10      }
11    }
12    Backend section {
13      *Pages {
14        *Page : Participants
15        ...
16    }
17  }
18 }
```

Listing 5.9: Extension Model for Conference Component

Listing 5.10 illustrates a more sophisticated page reference. The language allows the specification of how a module can access the data it handles. In the example, the module has a reference to the existing **participants** page. Additionally, the example specifies that the module should use the backend data access object (DAO) of the existing **MyConference** component instead of managing the database access itself. Besides this variant other options are possible. The **frontendDAO** option can be used, if the DAO of the frontend section has to be used. With **database**, it can be specified that the module shall use a component's data by direct access to its database tables. If the data should come from a web service, the **webservice** option can be used. The **from** keyword is used to initiate references to the existing component. Instead of a modelled component, it is possible to create a reference to the WCMS core features (*content*, *menu*, and *user*). This kind of mapping can also be used for page references in component specifications.

```

1 Module participants {
2   Manifest {...}
3   languages {...}
4   *Page : Participants from : MyConference data backendDAO
5 }
```

Listing 5.10: Extension Model for Conference Module

Plugin specifications must include an assignment to a specific plugin kind (**PluginType**). The selectable plugin kinds correspond to the existing ones supported by the Joomla WCMS. This should cover plugins of other WCMSs as well. So, further code generations can create appropriate plugin code. Some plugin kinds handle data of existing extensions. Search plugins, e.g., usually proceed a search on data of specific component data. An example for this can be found in Listing 5.11. The modelled plugin has to deal with the data of the participants entity.

```

1 Plugin Participants_Search {
2   Manifest {...}
3   languages {...}
4
5   pluginType = search
6   *entities = Participant
7 }

```

Listing 5.11: Extension Model for Conference Search Plugin

In addition to the previously described extension kinds, the eJSL DSL supports *libraries* in the extension part. **Library** definitions can contain references to existing entities (see Listing 5.12). Moreover, libraries can contain **classes** and **packages**. Packages in turn can contain other packages and classes, whereas classes may extend or have references to other classes. Within a class, modellers can specify **methods** with **methodParameters** and a **returnValue**. This part of the eJSL language is inspired by common OO implementations. This allows modellers to create code-related model elements for individual purposes. We forewent to provide language elements for the explicit specification of class attributes within a class definition. However, since classes can contain entity references, modellers can augment them by entity attributes. In other words, entities become classes extended by methods through class definitions.

```

1 Library MyConference {
2   Manifest {...}
3   languages {...}
4
5   *Entities Talk, Room
6
7   packages {
8     Package conference {
9       classes {
10        Class ParticipantHelper {
11          *entities = Participant
12          methods {
13            Method checkPayment {
14              returnValue = madePayment : Boolean Default = "false"
15              methodParameters {
16                methodParameter participantID : Integer
17            }
18          }
19        }
20      }
21    }
22  }
23 }

```

Listing 5.12: Extension Model for Conference Library

Additionally, eJSL also allows the specification of *templates*, the last supported extension type of popular WCMSs. Templates (or themes) are extensions with style definitions for a running WCMS instance (see Section 2.1.3). This feature is supported by the eJSL language by language elements for CSS style definitions as the model excerpt in Listing 5.13 exemplifies. Additionally, module positions can be specified. This allows to place modules or component views on a WCMS instance. The suitable position type can be assigned to the **PositionType** element.


```

1  Template myTemplate {
2    Manifest {...}
3    languages {...}
4
5    positions {
6      TemplatePosition name {
7        positionParameters {
8          PositionParameter name {
9            positionType = component
10         }}}}
11
12    cssblocks {
13      CssBlock ".myCSSClass" (
14        KeyValuePairs {
15          color = "#ff6347"
16          ...
17        })}

```

Listing 5.13: Extension Model for an Example Template

5.2.4 Core Support

In order to describe interactions between extensions and core features like content, users, and menus, within an extension model, the eJSL DSL includes a language feature for WCMS core support. To use this feature necessitates a suitable formal description of the core features.

In [183] a concept for integrating WCMS extension development and concrete WCMS instance development is presented (see Figure 5.10). This includes the interplay between an instance model of the eJSL language (e.g. a shop extension model) with the meta-model of WCMS instance features. So, actual instance models of WCMS instances (actual WCMS-based websites) can be described in combination with actual instances of a modelled extension. To this end, an additional meta-model for WCMS instances (*cJSL*) is defined, which includes the core features of a WCMS. This model incorporates the concepts which are presented by Saraiva et. al in [204, 205] and Trias et. al in [236]. These concepts propose MDE of concrete WCMS instances but provide limited support for the integration of custom extensions.

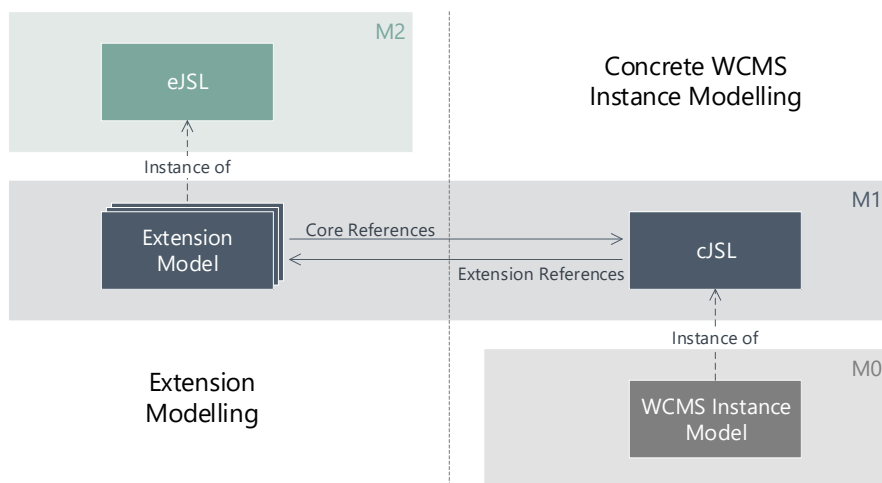


Figure 5.10: Concept of eJSL and cJSL Integration (cf. [183])

Based on the comparison of common WCMS features (see Section 2.1.3), the design of the cJSL language is inspired by the Joomla WCMS. However, the language concepts can be used for WCMSs in general, since their core features are fairly similar (cf. Section 2.1.2). In Figure 5.11, an excerpt of the cJSL meta-model is presented, whereas in Appendix B, a detailed overview of the language features is given.

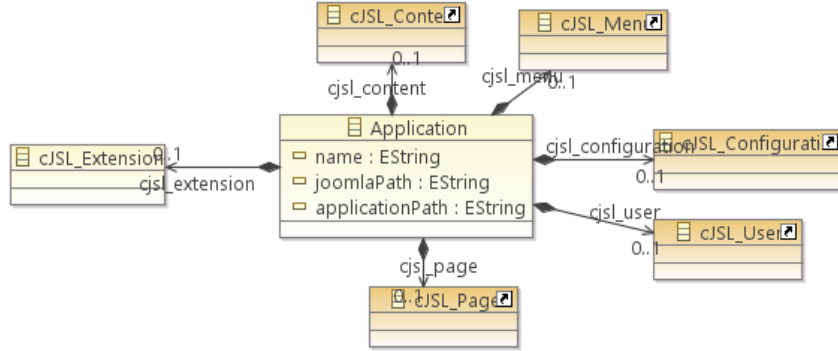


Figure 5.11: cJSL Meta-Model for Concrete WCMS Instances (Excerpt)

Due to the addressed research direction of this work, the focus has been led to extension development and maintenance. Therefore, the MDE concept of concrete WCMS instance development was not further researched. Though, in order to enable core support during extension modelling, the WCMS features of the cJSL meta-model should be accessible from an eJSL instance. To this end, we enriched the eJSL meta-model with features for WCMS core modelling. So, the same language can be used to create both *core models* and *extension models* which can access them directly without the need of model weaving techniques¹. We propose to incorporate the features of the cJSL meta-model into the core model. Though, extension developers can create own core models, based on their requirements. References to custom core models may not be supported by further code generation, though. Figure 5.12 illustrates the proposed core support based on the same meta-model, whereby the core model includes the features of a core meta-model for WCMS platforms (in our case the cJSL DSL).

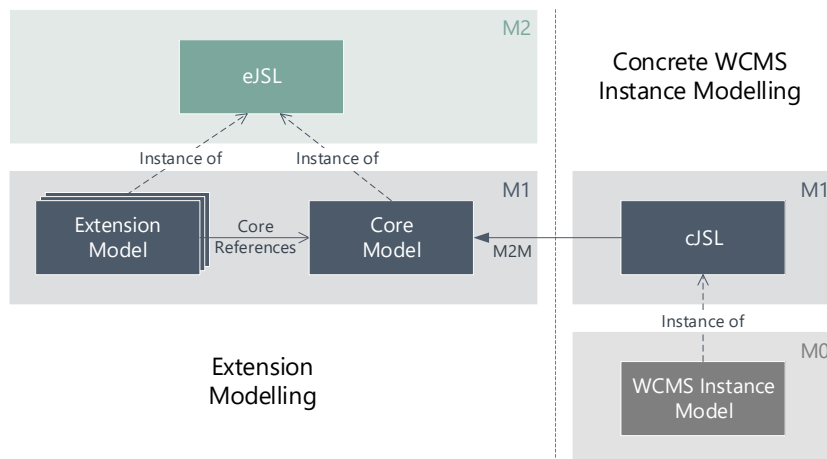


Figure 5.12: Core Support based on eJSL Models

¹Model weaving describes a generic operation that enables correspondences between model elements from different models based on different meta-models [53].

Listing 5.14 shows an example of a user management core model, which can be referenced by an extension model, as can be found in Listing 5.15. As part of future work, platform independent WCMS core feature definitions should be added to the DSL definition explicitly. Currently, the same language features are used for extension and core modelling. More specific, core models consist of entity and page definitions, which can be referenced by extension models.

```

1  eJSLModel "Core_User"{
2    eJSL part: CMS Core{
3      entities {
4        Entity usergroup {
5          attributes {
6            Attribute name {
7              type = Short_Text
8            }
9          }
10         Entity user {...}
11         Entity viewlevel {...}
12       }
13       pages {
14         CustomPage LoginPage {...}
15         CustomPage UserProfilePage {...}
16         CustomPage EditUserProfilePage {...}
17         CustomPage RegistrationPage {...}
18       }
19     }
20   }

```

Listing 5.14: Core Model for WCMS User Management

```

1  eJSLModel "Conference"{
2    eJSL part: CMS Extension{
3      entities {
4        Entity Talk {
5          attributes {
6            Attribute title {...}
7            Attribute ^description {...}
8            Attribute speaker {...}
9          }
10         references {
11           Reference {
12             EntityAttribute = speaker
13             ReferencedEntity = Core_User.user
14             ReferencedEntityAttribute = user.Name
15           }
16         }
17       }
18       pages {...}
19     }
20   }

```

Listing 5.15: Extension Model with Reference to the WCMS Core

5.3 Well-formedness Rules

In order to validate the consistency of extension models, we added a set of well-formedness rules for the eJSL DSL. These rules can be used to evaluate model instances and ensure valid model elements. In this sub-section, we present the implemented rules as formalized OCL specifications and describe them. In addition, we will explain some violation examples for a better understanding of constraints with higher complexity. In Appendix C a collection of all well-formedness rules as developed in the context of this work can be found.

The list of constraints for the eJSL language contains a set of similar validation rules for various contexts. The most straightforward rule concerns the identifiers of instance model elements. For the most elements the rule for unique identifiers follows a similar implementation as Listing 5.16 exemplarily illustrates for attribute identifiers in the *Entity* context. For all entity attributes, the constraint checks if the **name** property of attribute pairs which are not the same differ from each other. Additionally, the constraint considers the '^' prefix, if an identifier is used which is spelled like an eJSL keyword.

```

1 context Entity inv uniqueAttributeIdentifier :
2   self.attributes
3     ->collect( '^' .concat(name))
4     ->union( self.attributes->collect( name))
5     ->isUnique(a|a)

```

Listing 5.16: Constraint for Unique Attribute Identifiers (within one Entity)

5.3.1 Data Modelling

The set of constraints for the data modelling part of the modelling language includes various rules considering the well-formedness of entities, attributes, and references. Due to the supported inheritance of entities, some constraints are defined which ensure valid models.

To avoid that an entity inherits from itself, we implemented the constraint in Listing 5.17.

```

1 context Entity inv entityDoesNotExtendItself :
2   self.supertype -> excludes( self)

```

Listing 5.17: Constraint for consistent Entity Inheritance

To avoid possible generalization cycles, a more sophisticated problem of inheritance, the previously described constraint can be easily extended by the closure function. This function ensures a transitive closure of the whole inheritance hierarchy. So, even cycles in a more complex hierarchy will be found (Figure 5.13). Therefore, this constraint replaces the previously described rule as well, since a generalization cycle also exists, if an entity inherits from itself. In [47] and [14], the theoretical background for transitive closure of inheritance in modelling languages can be found.

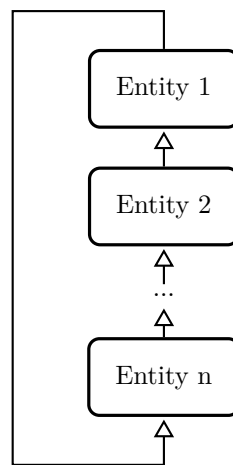


Figure 5.13: Generalization Cycle between Entities

```

1 context Entity inv noGeneralizationCycles :
2   self.supertype
3     -> closure(supertype)
4     -> excludes(self)

```

Listing 5.18: Constraint for Transitive Closure of Entity Generalization (avoid Generalization Cycle)

Moreover, rules for attribute types have been implemented. So, invalid or unnecessary type classifications can be identified. One of these rules limits the auto increment property of attributes to the **Integer** type. This constraint relates to the **StandardTypes** context which is a sub context of attributes.

```

1 context StandardTypes inv autoIncrementForInteger :
2   self.type=StandardTypeKinds::Integer
3   implies
4   self.autoincrement

```

Listing 5.19: Constraint for Auto Increment Property

In the context of entity references, a set of rules for valid multiplicities exist. Listing 5.20 shows an example for the valid range of multiplicity values, whereas Listing 5.21 illustrates the constraint for relations between the minimum and maximum value of a reference. The latter is required to exclude inaccurate multiplicities like a maximum of '0' or a minimum multiplicity, which is higher the maximum.

```

1 context Reference inv allowedMinValues :
2   Set{'0','1','-1'} -> includes(self.min)

```

Listing 5.20: Constraint for Multiplicity Values

```

1 context Reference inv consistentMultiplicities :
2   self.max <> '0'
3   and
4   (
5     self.min <> '-1'
6     or
7     self.min = self.max
8   )

```

Listing 5.21: Constraint for Valid Multiplicity Relations (Between min and max)

In addition, a rule for transitive closure of referenced entity attributes is defined (Listing 5.22). So, loops between references, as Figure 5.14 illustrates, can be avoided. The constraint checks, if no attribute in the closure of all referenced attributes references on the attribute of the current reference context. This constraint is relevant for complex models with a sophisticated entity section.

```

1 context Reference inv noReferenceCycles :
2   self.entity.references
3     -> closure(entity.references)
4     -> forAll(r|r.attributereferenced <> self.attribute)

```

Listing 5.22: Constraint to avoid Entity Reference Cycles

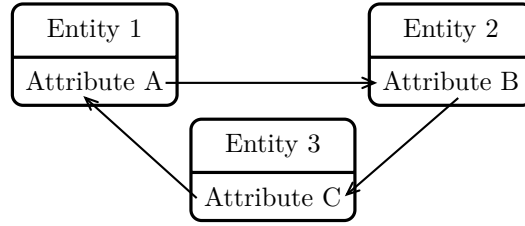


Figure 5.14: Reference Cycle between Entity Attributes

5.3.2 Interaction Modelling

In addition to the data modelling constraints, we propose a set of rules for the interaction modelling part of the language. This includes rules for all page kinds and comprising links to other pages.

The first set of constraints addresses the consistency between the field type of a details page field and the type of its mapped entity attribute. An example can be found in Listing 5.23. For edit fields of the HTML type `Datepicker`, the constraint checks, if the mapped entity attribute is of type `Time`, `Date`, or `Datetime`. In Appendix C, further similar rules for type consistency of detail page fields can be found.

```

1 context DetailsPage inv detailsPageFieldDatepicker :
2 let attrType : Sequence =
3   self.editfields.attribute.type.oclAsType(StandardTypes).type.oclAsType
4   (StandardTypeKinds)
5   in attrType = StandardTypeKinds::Time->asSequence()
6   or attrType = StandardTypeKinds::Date->asSequence()
7   or attrType = StandardTypeKinds::Datetime->asSequence()
8 implies
9 self.editfields.htmltype.oclAsType(HTMLTypes).oclAsType(SimpleHTMLTypes)
10 .htmltype.oclAsType(SimpleHTMLTypeKinds) =
11   SimpleHTMLTypeKinds::Datepicker->asSequence()
  
```

Listing 5.23: Constraint to Check if Datepicker (HTML Type) is Mapped to Time, Date and Datetime (Entity Type)

Another constraint (Listing 5.24) considers the consistency of a link in an index page which represents an entity. If an attribute is used as link, it must exist in the referenced entity and the linked details page must refer to the same entity (cf. Figure 5.15).

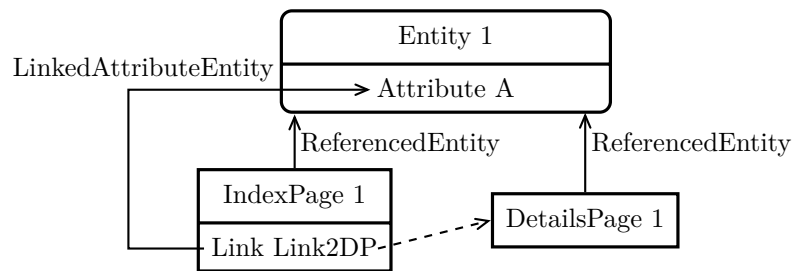


Figure 5.15: Link Attribute Must Exist in Referenced Entity

```

1 context IndexPage inv linkedIndexAttributeConsistent :
2   self.links
3   -> forAll(
4     l | self.entities.attributes.name->includes(l.linkedAttribute.name)
5     and
6     self.tablecolumns.name
7     -> includes(l.linkedAttribute.name)
8   )

```

Listing 5.24: Constraint to Check if Linked Attribute in IndexPage is Consistent to Referenced Entity Attribute

In Listing 5.25 we present a rule to check if table columns in index pages are consistent to the referenced entity. I.e., the table columns must be consisting attributes of the referenced entities.

```

1 context IndexPage inv RepColsandFiltersConsistent :
2   self.tablecolumns
3   -> forAll(
4     tc | self.entities
5     -> includes(tc.oclContainer())
6     and
7     tc.oclContainer().oclAsType(Entity).attributes
8     -> includes(tc)
9   )

```

Listing 5.25: Constraint to Check if Table Columns are Consistent to Referenced Entity in IndexPage

An additional check is used as consistency check for multiple referenced entities within a page definition (see Listing 5.26). The DSL allows to define a set of multiple referenced entities which must consist of references to each other. So, related attributes of different entities can be joined in one page. A valid model may consist of tree-based entity relationships with one base entity, since reference cycles are not allowed (see Listing 5.22). Therefore, the constraint fails, if no or more than one base entity exists.

```

1 context IndexPage inv MultiplePageEntityReferences :
2   let mainEntity : Sequence(Entity) =
3     self.entities
4     -> select(
5       e | self.entities.references.entity
6       -> excludes(e)
7     )
8   in entities->size() > 1
9   implies mainEntity->size() = 1

```

Listing 5.26: Constraint to Check the Consistency of Multiple Referenced Entities

5.3.3 Extension Modelling

The extension modelling constraints mainly consist of checks for unique extension names, language keys, and library class and method identifiers. Unique extension names, however are only necessary for the same extension kind (see Listing 5.27). Other necessary checks, like correct use of sections or references between extensions, are covered by the automatically generated validation rules based on the grammar definition.

```

1 context CMSExtension inv uniqueExtNames :
2   self.extensions.ocltypes()
3   ->forall(
4     t | self.extensions->select(ocltypes() = t)
5     ->isUnique(name)
6   )

```

Listing 5.27: Constraint to Check Unique Extension Names (Same Extension Kind)

5.4 Model Editors

To use the eJSL DSL, a set of plug-ins for the most commonly used development environments in the WCMS domain is provided. So, we avoid the problem of missing supportive tools during extension development (cf. problem statement 5). Based on own experience and the requirements of the WCMS community, these environments currently are *IntelliJ IDEA*, *PhpStorm*, and *Eclipse*. The provided editor plug-ins are customized for integration with each of these environments. All IDE plug-ins are available on GitHub and can be easily installed directly to the preferred IDE via the respective plugin repositories: <https://github.com/thm-mni-ii/JooMDD/tree/master/updateSites>. The IDE plug-ins provide a textual editor with syntax highlighting, error messages, dependency checks, and auto completion support for keywords and references between model elements. Figure 5.16 illustrates the plug-in representation within Eclipse, whereas Figure 5.17 shows the plug-in within PhpStorm.

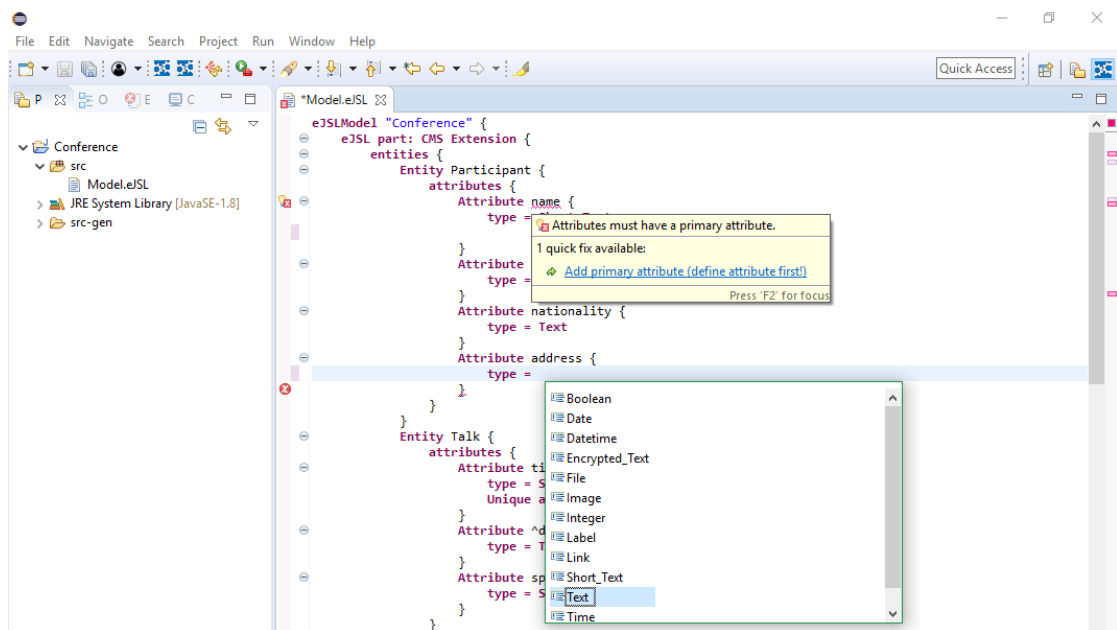


Figure 5.16: eJSL Model Editor within Eclipse IDE

During model-driven engineering of WCMS extensions, developers typically expect kinds of supportive project initialization dialogues to avoid modelling from scratch (cf. Section 4.1). Therefore, we incorporated a project wizard to the editor plug-ins (see Figure 5.18). This wizard can be used for the initial creation of the eJSL project structure and allows to choose one of our showcase models (see Section 5.5 below) as initial example model. This feature proved to be worthwhile during the application of MDE during WCMS extension development (see Chapter 7).

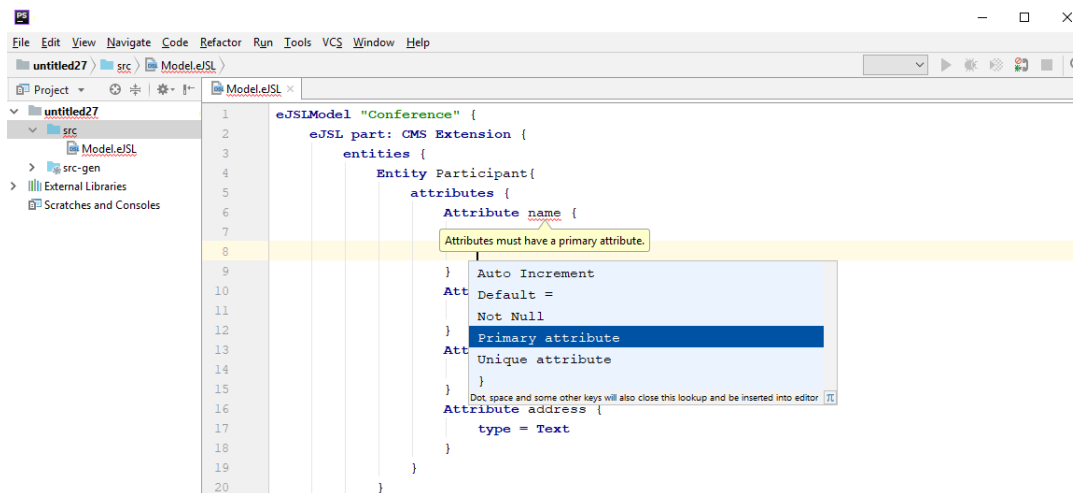


Figure 5.17: eJSL Model Editor within PhpStorm IDE

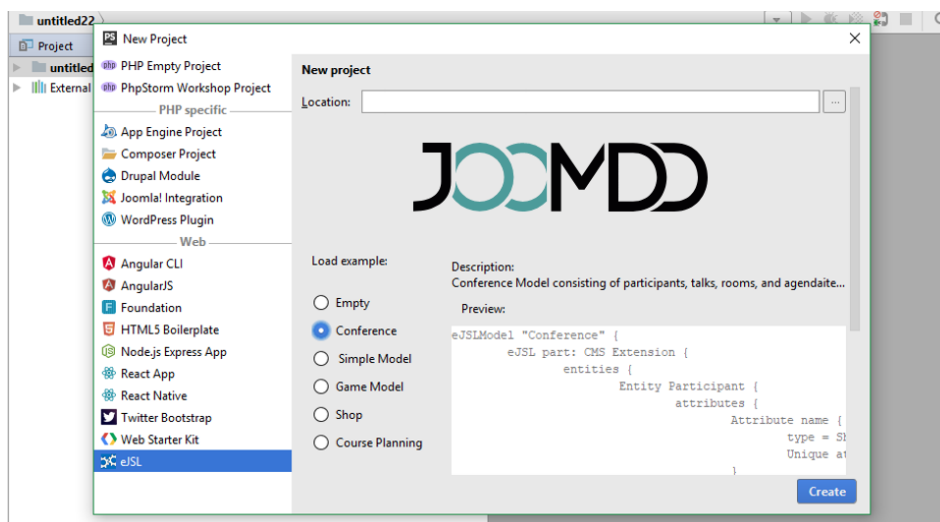


Figure 5.18: Project Wizard within PhpStorm IDE

In addition to the IDE plug-ins, a platform-independent web IDE is provided (see Figure 5.19): <https://tinyurl.com/joomdd-web>. This web IDE facilitates eJSL model editors with Joomla-specific transformation tools (see Chapter 6). So, developers can apply MDE of WCMS extensions without installing the plug-ins locally. All the features of the editors in the plug-ins are also provided by the web-based editor.

5.5 Showcase Models

In order to support extension developers during the use of the eJSL language, a set of showcase models are provided. These showcase models comprise examples of various complexity, providing a steep learning curve for new developers. In this section we present some concepts of the showcase models, whereas the actual eJSL models can be found publicly available on GitHub: <https://github.com/thm-mni-ii/JoomDD/tree/master/de.thm.icampus.joomdd.ejsl.parent/de.thm.icampus.joomdd.ejsl/instances>.

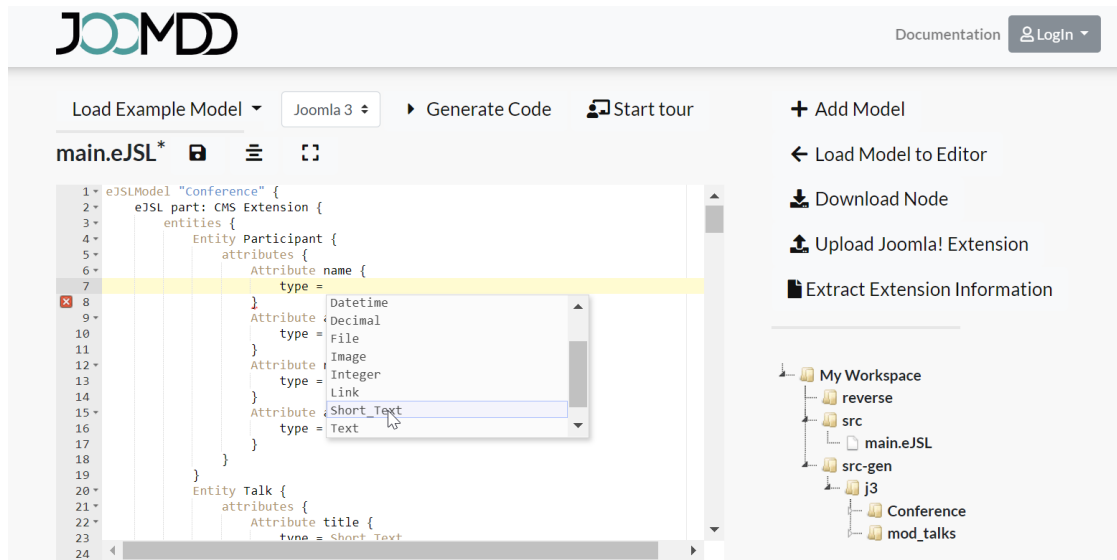


Figure 5.19: Web IDE Comprising the eJSL Model Editor

Conference and Pre-Course Models

The *conference model* which is also used as running example in this work, incorporates a set of straightforward WCMS extensions, including a component with a set of 8 frontend and 8 backend views, as well as a dependent module which illustrates data from the component. The model comprises 4 entities (participants, talks, rooms, and programme) with appropriate references to each other. Additionally, the model contains 8 page definitions - one index and one details page for each entity including links to each other. Figure 5.20 illustrates the references between the entities, pages and extensions of the conference model. The conference model was also used as basic example during the controlled experiment (see Chapter 7) in order to create extensions with standard CRUD views.

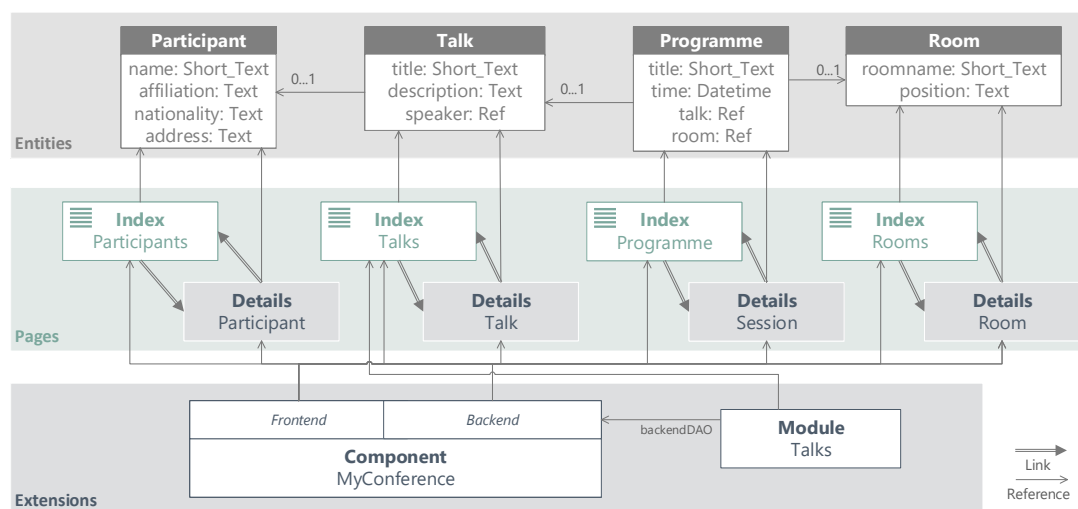


Figure 5.20: Showcase Model - Conference

A similar showcase model which can be used as reference model is represented by the *pre-course model*. This model was initially created during the model-driven engineering of a Joomla extension for pre-course management at the university of applied sciences in Gießen, Germany (Technische Hochschule Mittelhessen, THM). This and other use cases are presented in Section 6.5.1 below. The pre-course model consists of 4 entities and 8 pages which are referenced all by the backend and frontend section of a component in the extension section. In contrast to the conference model, the pre-course model does not include a module definition. The references between the pages and entities are similar, though.

Shop Model

In contrast to the previously described conference and pre-course models, the *shop model* comprises a more complex model structure, based on various extension kinds.

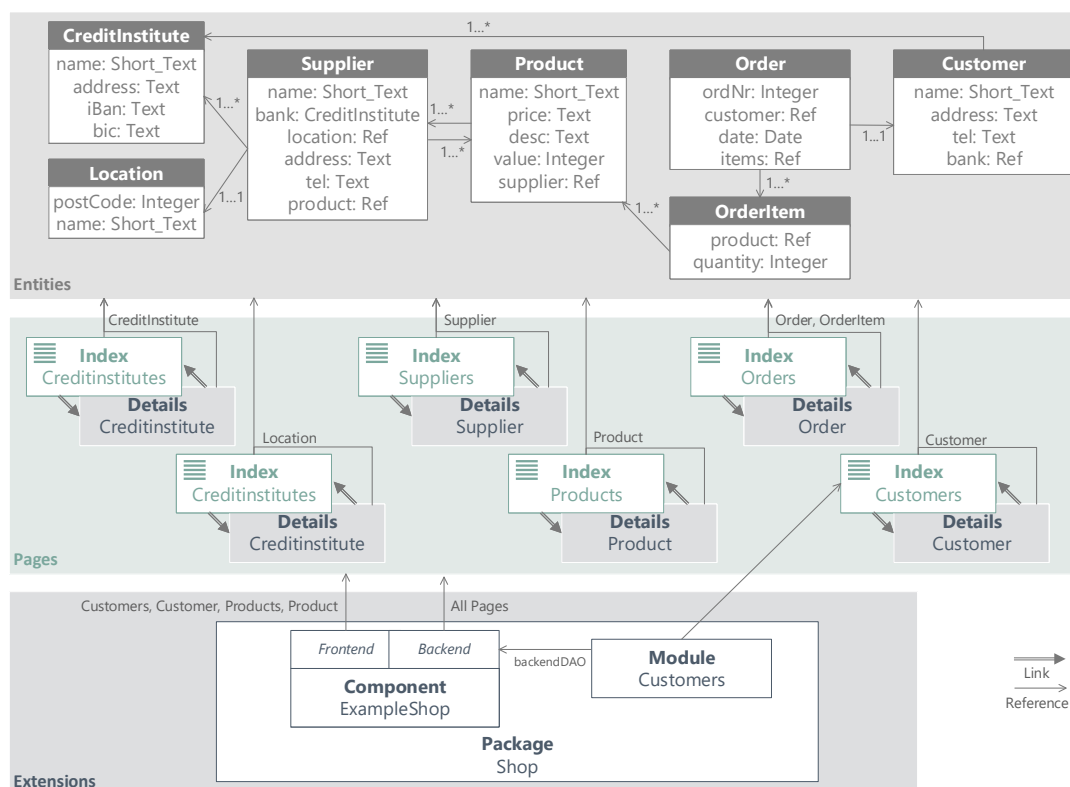


Figure 5.21: Showcase Model - Shop

The shop model consists of 7 entities and 12 pages in total. Similar to the previously described models, one index and one detail page for each entity is defined. The only exception is given in the **orders** and **order** page definitions. These pages refer to the **Order** and **OrderItem** entity, which form a set of belonging entities. The speciality of the shop model can be found in the extension definition. Similar to the conference model, the shop model contains a component and dependent module specification. Though, both extensions are specified as part of a package extension. This information may be helpful during code generation, since implicit dependencies between the extensions can be assumed.

Weblinks Model

The *weblinks model* comprises similar features, such as the weblinks component [181] for the Joomla WCMS. This component was part of the Joomla WCMS core until 2015 and is now one of the independent extensions, which is maintained by the Joomla community. This component is often used as reference by extension developers, since it adopts the typical features of a Joomla component and follows the coding guidelines of the community (cf. Section 2.2.1). The model has one entity and 4 pages (2 index and 2 detail pages) but the most detailed parameters and language definition. Moreover, a component as well as a dependent module is specified. The model was created automatically by the model extraction tool, which is presented in Section 6.4.

Generic Example Model

In addition to the previously described model examples, a generic model example is provided. In order to allow a straightforward creation of own models, the simple model consists of model elements with placeholders. The complexity of the model is similar to the weblinks model. Listing 5.28 shows an excerpt of the pages section which shows the format of the placeholder identifiers which are used in the example model.

```

1  IndexPage IndexPage2 {
2      *entities = entity2
3      representationColumns = entity2.attribute1 , entity2.attribute2
4      filters = entity2.attribute1 , entity2.attribute2
5      links {
6          InternalLink details {
7              target = DetailsPage2
8              linkedAttribute = entity2.attribute1
9          }
10     }
11 }
12
13 DetailsPage DetailsPage2 {
14     *entities = entity2
15     links {
16         InternalLink allEntries {
17             target = IndexPage2
18             linkedAttribute = entity2.attribute1
19         }
20     }
21 }
```

Listing 5.28: Generic Showcase Model with Placeholders (Excerpt)

5.6 Evaluation

In order to evaluate the previously presented DSL and corresponding editors for abstract WCMS extension specifications, we discuss the *coverage of the requirements as defined in Section 4.3* above as well as the *adequacy of the language with regard to existing design guidelines for DSLs*. Moreover, we discuss the validity threats of the design decisions. Since the language is designed for model creation as basis for code generation, we cover empirical evaluation by case studies using the DSL and its editors in combination with suitable transformation tools within the next chapter (Chapter 6).

5.6.1 Requirement Verification and Validation

In this sub-section, we examine the coverage of the DSL and editor requirements as defined in Section 4.3 by our presented artefacts. We check, whether the acceptance criteria for each of the related requirements is fulfilled or not. This is in accordance with the verification and validation definitions of Boehm in [26]. In order to address the respective requirement, we use the defined requirement identifiers for the DSL (R1.1-R1.6) and corresponding model editors (R2.1-R2.3) as well as the general infrastructure requirements (R5.1-R5.3) as defined in Table 4.7 in Chapter 4.

Subject: Domain-Specific Language

The presented language provides appropriate features for data modelling, within a respective **entities** section in a model instance, in order to fulfil Requirement R1.1. This includes the possibility of defining entities with attributes and references to other entities. Additionally, inheritance and the specification of own attribute types are provided. The current version of the DSL allows data modelling independent on the domain or a specific WCMS. This allows to use the language in various contexts and domains, even though we propose to use the language for extension definition in the WCMS domain.

With the **pages** part of the presented language, representations of defined entities can be specified. To this end, the DSL provides various representation kinds for index, details, static, and custom pages. These may include custom definitions, e.g. for the entity fields which have to be represented. Additionally, interactions between pages in form of various link types can be specified as well as individual actions based on page elements. These features address and fulfil the acceptance criteria for Requirement R1.2.

By providing the **extensions** modelling section, the DSL allows modellers to encapsulate modelled pages in form of WCMS extensions. According to Requirement R1.3, abstract modelling elements for extension meta data and supported languages is provided. The latter can be explicitly extended by custom translations for specified languages. In order to specify references between different extensions, the language provides suitable language features. Even though the eJSL language provides various extension kinds, inspired by the sophisticated extension mechanism of Joomla, the extension modelling part is abstract enough to allow specifications independent of the targeted WCMS.

To cover Requirement R1.4, the DSL provides an interface to create references to WCMS core features (user, menu, content) within the extensions section. This allows the definition of dependencies between deployed extensions, such as a new module which extends a core component of the Joomla WCMS. However, the restriction to the extension part is a current limitation of the language which has to be extended to the entities and pages part within future DSL refinements. So, dependencies to the core can be specified explicitly for extension parts like models or views.

In accordance to Requirement R1.5, the DSL allows **@preserve** annotations within instance models for entities, entity attributes and references, and all kinds of pages. This annotation indicates preservable model elements, shall be ignored during transformation processes like code compilation. So, development scenarios which consider extensions as dependencies which do not have to be generated can be realised. Additionally, the development of partial extensions for already deployed extensions can be covered, if adequate code generators use this information.

With the specification of well-formedness rules for instances of the eJSL DSL, we fulfil Requirement R1.6. These rules assure the quality of defined extension models during the modelling process. So, modellers can avoid errors during the code generation process and inappropriate extension code. A variety of well-formedness rules are defined in the OCL syntax. So, MDE infrastructure developers can transfer the rules into appropriate formats, e.g. for validation or refactoring functionality within corresponding model editors.

Subject: Model Editors

The presented model editors allow the creation of text-based instance models of the eJSL language, including all specified language features (Requirement R2.1). In order to support extension developers during the modelling process (Requirement R2.2), the introduced editors provide syntax highlighting, auto completion, and error validation. To ensure readable models after automatic reverse engineering steps, the editors also provide model formatters.

Since we use the Xtext framework for DSL development, the basis for adequate textual editors, for both the Eclipse IDE and IntelliJ IDEA, can be generated automatically. These generated editors already include rudimentary validators, syntax highlighting, and auto completion which are refined and extended by the previously mentioned formatters. Additionally, an editor plugin for the PhpStorm IDE and a completely platform-independent, web-based cloud IDE are provided. The latter incorporates both the DSL-specific editor features for user support as well as the transformation tools which are explained in the next chapter (Chapter 6). The additional editors are also incorporated into an automatic build process in order to cover Requirement R5.3. Moreover, quality assurance (Requirement R5.1) for the editors and showcase models is ensured by language tests which fail, if DSL refinements are not covered by the showcase models or editor features which have been additionally added to automatically generated editor artefacts. The editor features which are automatically generated by the Xtext framework meet the quality assurance requirement implicitly, since they are directly generated based on the DSL. So, every refinement is considered during a re-build of the editors.

As indicated by Kahani et al. in [121], working with Eclipse-based MDE tools can become a challenging task - especially for new MDE adopters. Therefore, the automatically generated Eclipse based-editors plugins are extended by plugins for other common IDEs used in the WCMS domain such as PhpStorm, as well as a complete platform-independent cloud IDE. So, extension developer must not deal with a completely new development environment. The variety of editor plugins addresses the platform-independence (Requirement R2.3) and enables a straightforward integration into the preferred development process of extension developers (R5.2). The latter is also supported by project wizards and the presented showcase models, which provide developers a starting point for extension development.

5.6.2 Adequacy of the DSL

As discussed in the previous subsection, the presented DSL and editors are developed in accordance to the specified MDE infrastructure requirements from Section 4.3. However, the acceptance criteria is expressed on a high level due to the applied iterative development process. Therefore, we discuss our language design decisions based on published guidelines such as the ones presented in [8] and [127]. In [8], the authors propose a comprehensive process for framework-specific language development. Even though the guidelines are well-described with case studies, it would go beyond the bounds of this work to follow the complete guidelines. Though, the proposed development process is similar to the followed iterative process of this work, which is driven by the common development scenarios in the domain (Section 2.3). Karsai et al. propose a lean set of guidelines for DSL development in [127], based on the following categories: *Language Purpose*, *Language Realization*, *Language Content*, *Concrete Syntax*, and *Abstract Syntax*. Based on these categories, we reflect on the language design of our DSL hereinafter.

Language Purpose: The proposed language will be used for code generation and model discovery of WCMS extensions in order to cover the identified development scenarios. Extension features are considered on a high abstraction level. So, the language can be used independently for any target WCMS. The features are elicited from reference extensions and interviews with industrial practitioners. These developers are the target group which is intended to use the language.

Language Realization: Due to the specialised requirements of the problem domain, the language was implemented from scratch. Though, existing useful concepts from other languages have been incorporated. Based on the targeted modellers, we decided for a textual realisation based on Xtext. However, a graphical concrete syntax can be created on the basis of the defined DSL. By using Xtext, the abstract syntax of the language is generated in the form of an ecore model, which may serve as basis for a GMF-based editor. The well-formedness rules were specified in OCL, allowing to embed them into the language and corresponding editors. By using a well-known type system as basis in the language allows extension developers a straightforward adoption.

Language Content: To keep the language as simple as possible, we decided to add necessary domain concepts iteratively, based on the reference extensions and requirements from actual developers. So, it is possible that the language is missing some special features which are supported by the domain but not necessarily used by developers. We limited the number of language elements and possible constellations by our well-formedness rules. Additionally, only a few elements must be mandatory in an instance model. We researched our reference extensions carefully during the decision of putting an extension feature into the DSL or corresponding transformation tools. E.g., the DSL does not reflect the extension architecture of a specific WCMS. This technological knowledge must be placed in code generators for a specific WCMS (cf. Section 6.2.2).

Concrete Syntax: The textual concrete syntax of the eJSL DSL adopts the common notation of extension developers who are used to write program code in web languages like PHP and JavaScript. To this end, a simple descriptive notation with curly brackets for scope borders, sets of key-value pairs, and symbols for the definition of comments is provided. These syntax elements allow the specification of hierarchical structures and can also be learned even by less experienced developers. One difference, compared to common programming languages in the domain, is that one symbol (=) is used for both declarations and assignments. This feature, however, was not criticised during the tutorials or experiments. By using the same notation style throughout the whole language for various features, we intended to increase the understandability and make the language more intuitively usable. All identifiers can be specified with qualified names, to provide references between model elements of different models organized in different files. Explicit import and export features are not supported.

Abstract Syntax: Most of the elements in the abstract syntax align to the concrete syntax in order to keep internal transformations and model presentation as simple as possible. To this end, similar concepts are reused through sub-classing, e.g. for **page** and **extension** kinds, or all kinds of **types** which are extendable by custom types in an instance model. Moreover, the composite pattern is realised for extension kinds in order to extend them easily during further refinements. The composition of the language into **entities**, **pages**, and **extensions** enables modularity and increases the flexibility during modelling.

5.6.3 Threats to Validity

The presented DSL for WCMS extensions and the corresponding text-based editors were developed in accordance to the specified requirements as defined in Section 4.3. These requirements resulted from the experience of extension developers for the Joomla WCMS. However, external extension developers were not involved during the evaluation of the resulting artefacts after each iteration. This is a threat to *internal validity*, since design decision which were implemented in earlier versions could have affected the current state of the artefacts. In Chapter 7, we present the results and lessons learned of a hands-on tutorial with external developers who used a late version of the DSL in addition to Joomla-specific transformation tools. Some of the results indicate suboptimal design decisions which could have been avoided, if additional experts were involved earlier.

A threat to *external validity* is based on the requirements which are derived from the experience of developers from the Joomla community, as well as the feature comparison of the most popular WCMSs (WordPress, Joomla, and Drupal). Further research has to ensure that the language is also suitable for other less popular WCMSs like TYPO3, Shopware, or Contentful. Moreover, the user experience evaluation of the language and the corresponding editors is currently limited to extension developers from the Joomla community, based on the empirical assessments which were conducted. Event though the language is designed platform-independently, it has to be further researched, if it is also suitable for other WCMSs with high popularity.

If forward engineering is about moving from high-level views of requirements and models towards concrete realizations, then reverse engineering is about going backwards from some concrete realization to more abstract models [...]

– Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz in [54]

In addition to the previously described modelling language for WCMS extensions, we introduce a set of transformation tools in the context of the Joomla WCMS in this chapter. These tools complete the MDE infrastructure to perform forward and reverse engineering steps in order to apply the common WCMS extension development scenarios in a model-driven manner as proposed in Section 4.2. This includes the development of new independent and dependent extensions as well as the migration and modernization of legacy extensions. So, the problem statements 1-5 as presented in Chapter 1 are addressed. We choose Joomla as target WCMS, since it provides the most sophisticated extension mechanism, compared to the other market leaders (cf. Section 2.2.2). So, starting with Joomla as first target platform, implementations for other WCMSs should be possible with less effort as part of further work. We discuss related work and highlight suitable approaches which were considered during tool development. Moreover, we introduce general concepts and Joomla-specific implementations of both code generators (*model-to-code transformation*) and reverse engineering facilities (*code-to-model transformation*). The presented generators address the challenges which occur during the development and migration of Joomla extensions (cf. problem statements 1 and 2), whereas the reverse engineering facilities address the challenges which originate from dealing with legacy extensions (cf. problem statements 3 and 4). The platform-specific implementations serve as proof of concept and can be directly used during empirical studies such as the ones presented in this work (cf. Section 7.3 in Chapter 7).

The presented tools are incorporated with the previously described DSL and corresponding editors as complete MDE infrastructure for extension development targeting the Joomla WCMS. The infrastructure project, called **JooMDD**, is publicly available on GitHub: <https://github.com/thm-mni-ii/JooMDD>.

6.1 State of the Art

As already mentioned in Chapter 5, several MDE approaches in the web domain exist, but do not cover the domain-specific requirements in order to develop WCMS extensions. For example, the code generation approach as proposed by Saraiva et al. in [206] covers concrete WCMS instances but does not consider extensibility scenarios through standardised extension types taking their interdependencies into account. However, some of these works provide adequate transformation approaches which are highlighted within this section. Moreover, we compare existing WCMS-specific MDE infrastructures, based on the development scenarios and problem statements which are stressed in this work. This includes general WCMS-specific generators for model-to-code transformations and reverse engineering facilities for code-to-model transformations. Additionally, we research existing Joomla-specific transformation tools and determine their adequacy for our MDE approach.

6.1.1 Translation to Extension Code

In accordance to Jörjes [114], code generation is the key factor for the application of an MDE approach, treating models as primary development artefact. As described in Chapter 3, code generators enable model-to-code transformations based on different implementations such as template-based generators. The development of code generators requires the elicitation of repetitive code parts which e.g. are put into generator templates. In [45] the authors present a technique for the automatic examination of existing code in order to identify schematically redundant code parts. A consideration of this aspect in this work will be a valuable contribution in order to minimize the effort during generator development.

Based on the results of our conducted pre-study during the examination of reference extensions for the Joomla WCMS, almost 70% of the researched views are characterised by their provided CRUD functionality (cf. Section 4.1). This is a promising aspect which may be addressed by automatic code generation in order to reduce development effort during extension development (cf. problem statement 1 and 2). Papotti et al. evaluate the efficiency improvement during CRUD-intensive web application development in [177]. The authors conducted an experiment in a university environment and observed a time reduction of 90.89% by CRUD code generation.

In [194], the authors examine the CRUD aspect in web applications rigorously and propose an approach for the automatic extension of CRUD operations in IFML models. Though, since we decided to keep the DSL for WCMSs as abstract as possible, explicit CRUD definitions are not provided on model level. Therefore, in contrast to this work, we presuppose that according code generators implement this feature to implicitly generate CRUD operations based on a representation kind in the instance model (cf. Section 5.2.2). The automatic generation of code for CRUD-intensive web applications has been addressed by various works, mostly in form of scaffolding generators. These generators can be used during early stages in development, but do rarely adopt an MDE approach in order to support developers throughout the whole development life cycle. Typically, these approaches operate at a low abstraction level and require manual adaptations of generated artefacts.

As previously mentioned, the MDE infrastructure presented in [66] is the only work addressing MDE of WCMS extensions more extensively. The proposed concepts are intended to cover extension development platform-independently. However, the authors evaluate their proposed concepts with code generators on the basis of the DotNetNuke WCMS, which has a significantly less market share than WordPress, Joomla, or Drupal. Moreover, the adequacy of the approach is not evaluated for WCMSs with a more sophisticated extension mechanisms, such as provided by the Joomla WCMS. Nevertheless, we take concepts like e.g. the smart and dummy approach [66, 193] during the development of code generators for MDE support into account.

Existing platform-specific code generators in the WCMS domain are limited to initial development scenarios for independent and corresponding dependent extensions (cf. problem statement 1, 2, and 4). Table 6.1 summarizes existing tool support for the stressed development scenarios of this work (cf. Section 4.2), exemplarily for the Joomla WCMS (version 3).

Table 6.1: Tool support for WCMS Extension Development Scenarios (for Joomla 3)

Tool	S.1	S.2	S.3	S.4	S.5
Component Generator [217]	✓	✗	✗	✗	✗
Joomla Component Builder [242]	✓	✗	✗	✗	✗
Component Creator [102]	✓	✓	✗	✗	✗
Component Architect [214]	✓	✓	✗	✗	✗
JCCreator [4]	✓	✓	✗	✗	✗

Even though initial development scenarios are covered (cf. scenario 1), none of the frameworks provide partial extension augmentation (cf. scenario 2 and 4), or support extension developers during extension migration (cf. scenario 3) or modernization activities (cf. scenario 5). Nevertheless, we consider existing implementation details of these works during the implementation of more sophisticated code generators for WCMS extensions below.

None of the existing approaches, presented in this section, can be used as direct MDE solution for the specified problems during development scenarios 1-5 which require model-to-code transformations. The discussed related work either addresses WCMS instances with their extensions, only supports WCMSs with a less sophisticated extension mechanism, or exclusively supports developers during development scenario 1 (e.g. for the exclusive generation of CRUD code). Therefore, we present concepts and platform-specific implementations for model-to-code transformations by code generators taking all the presented WCMS extension development scenarios and related problem statements into account.

6.1.2 Handling of Legacy Extensions

In order to apply augmentation, modernization, or migration scenarios in a model-driven manner, appropriate reverse engineering approaches for existing software artefacts must be realised. So, the challenges during these scenarios can be addressed. This includes tedious reverse engineering processes required for migration or augmentation of legacy extensions (cf. problem statement 3), as well as management of dependencies between extensions (cf. problem statement 4). Various existing approaches address reverse engineering in a model-driven context (MDRE). Most of the approaches provide general concepts for model extraction based on existing source code, whereas only a few works consider MDRE in the context of WCMSs.

A general approach for automatic extraction of models based on source code is presented in [6]. The authors propose and evaluate automatically retrieving of framework-specific models, based on a simple code analysis technique. Due on the promising results, we follow a similar approach in order to apply model discovery of WCMS extensions. However, further model transformation techniques are required, since the abstraction level of resulting models is too low to fit to the previously described WCMS-specific DSL.

Frameworks such as MoDisco [29] and MEFiSTo [82], provide solutions for MDRE from source code to various abstraction levels including code-to-model and model-to-model transformations. The *MoDisco* framework provides an environment for tool developers in order to develop solutions for various MDRE scenarios. The framework facilitates a set of Eclipse plug-ins based on EMF and supports several OMG standards [29]. This includes meta-models, model discoverers, and a set of use-cases. A speciality of MoDisco is given by its extendibility by custom model discoverers in form of code parsers. This feature led to a variety of discoverers for various systems. However, most of the discoverers cover Java-based applications, whereas the implementation of web-based languages, such as PHP, received less attention. Existing Java-based parsers for the PHP language are outdated and can therefore not be used within custom discoverers. Therefore, the framework can not be directly used in order to fulfil the model extraction requirement (cf. Requirement 4.3.4) of this work which necessitates suitable discoverers for web-specific languages such as PHP and JavaScript.

With the *MEFiSTo* framework [82], tool developers can specify domain-specific MDRE solutions based on standardised meta-models such as the Knowledge Discovery Metamodel (KDM) [163]. In contrast to MoDisco, MEFiSTo addresses the development of modernization scenarios based on Situational Method Engineering [89]. The framework allows the flexible definition of MDRE processes, but requires a set of corresponding tools which are not provided by the framework. These tools, such as parsers and code generators, have to be implemented by tool developers in order to apply specified modernization scenarios. Another drawback of the framework is its

limitation to the KDM standard. MDE adoptions which are not based on this standard are not supported. This also includes the proposed approach of this work (cf. Section 4.2) which is based on the eJSL modelling language (cf. Chapter 5).

Another promising approach in the context of this work is represented by the *Gra2MoL* language [40]. This rule-based transformation language serves as interface between grammarware and meta-models (cf. Section 4.2). The language is intended to retrieve information out of source code and transform it to instances of abstract meta-models. In contrast to MoDisco and MEFiSTo, which require external model transformation tools, Gra2MoL is based on a rule sets which serves as layer between the grammar of the input code and the meta-model of the target instance model. So, the approach is usable for legacy code of any GPL. This makes the language useful during the development of model discoverers, also for MoDisco. Gra2MoL is available as part of an Eclipse plugin which is not updated since 2014 [39]. Unfortunately, the plugin is not compatible to newer IDE versions of Eclipse. Though, we will consider the concepts of Gra2MoL, as described in [40], during the development of adequate model extraction facilities for WCMS extensions.

Existing MDRE approaches in the web domain are tailored to specific frameworks, modelling language, or aspect such as the approaches described in [128] and [203]. The authors of [128] consider automatic model extraction based on ASP.net applications and a transformation to WebML [42, 41], whereas the proposed MDRE process in [203] addresses the examination of graphical user interfaces in order to extract model information. Even though the latter work is relevant for our work, since we have to retrieve UI information from legacy extensions such as extension view or widget representations (scenario 2-5), these works are not suitable in our specific domain.

MDRE principles have also been applied to address augmentation issues in the WCMS domain. Trias et al. introduce a reengineering method [237] and a reverse engineering tool for the migration of complete WCMS-based applications [238]. So, migrations of an instance from one WCMS to another WCMS can be realised. Even though the followed method can potentially improve the model extraction step based on our concept (cf. Section 4.2), its usability was presented by case studies for simple widgets for WordPress and Drupal, WCMSs with limited extensibility features. The usefulness for other WCMSs with a more sophisticated extension mechanism, such as Joomla, has yet to be investigated. We will incorporate the proposed method within our concept definition in this chapter, though. In [246], Vermolen et al. present an approach for the evolution of data models. This approach provides a well-defined strategy to deal with changes to existing data entities. Incorporating it into our work will help us to improve the extensibility during the (partial) augmentation of existing legacy extensions (scenario 2 and 4).

The previously described related work in the context of reverse engineering of legacy WCMS extensions is not directly adoptable to our specific problem statements. The abstraction level of resulting models created by most of the existing model discoverers is too low. Another open problem is, that most of the proposed model discoverers in the web domain are outdated or exclusively tailored to Java-based applications. Existing parsers for common web-based languages need to be implemented by tool developers in order to address reverse engineering challenges (cf. problem statement 2 and 3) related to modernization (scenario 3 and 5) or augmentation scenarios (scenario 2 and 4).

6.2 Code Generation of WCMS Extensions

In this section, we present the design decisions for the implementation of WCMS-specific code generators, based on the eJSL DSL which was presented in the last chapter (Chapter 5). To this end, we introduce a *general generator architecture* based on adequate architectural patterns. Moreover, a *prototypical implementation* of this architecture is demonstrated. This implementation comprises of extension generators for two major versions (3 and 4) of the Joomla WCMS.

6.2.1 Concept

Implementing a code generator for a specific domain is a complex task, since no common architectures or implementation methodologies are proposed. Usually, generators implement monoliths, tailored to a specific language or system. This leads to the following challenges during generator development and maintenance as stated in [62]:

- For each generator that is added (i.e. each new supported platform), redundant decisions regarding design and architecture have to be made.
- Adding new generators is also burdensome.
- Supporting a high number of generators obviously earns no “scientific merits” but it is required for business acceptance.
- Maintaining generators has an almost linear effort with regard to the number of generators.
- Platforms often provide best-practices for the architecture of apps. However, these best practices do not incorporate the peculiarities of the code generators.

To tackle these challenges, we propose a reusable and extendible generator architecture. This architecture realises various common design patterns¹. So, we can ensure more quality in generator implementations and simplify further adjustments. Moreover, our proposed architecture supports the extension of the generator in order to implement a cross-platform approach by variability, based on the template-based approach (cf. [81]).

In accordance with the statements of [197], it is common practice to *divide a generator into a front-end and back-end*. Thereby, the front-end part takes over the part of language processing, i.e. parsing input models and creating an internal representation. The latter then can be used by the back-end, the actual code generator. We follow these guidelines in order to split the generator architecture into a front-end and a back-end part with platform-independent and platform-specific components. So, we avoid a monolithic architecture and increase the reusability of the platform-independent parts of the generator.

Generator Front-End

Since the eJSL grammar has been defined with the Xtext framework, corresponding application code is automatically generated. This code serves as *model API* in Java applications and can be directly used within generator templates to access model information of eJSL model instances. The generated API code provides model elements as structured as in the Xtext grammar. However, the logic for sophisticated relationships between model elements has to be implemented by the generator itself. Therefore, we propose a *decorator pattern* to extend the automatically generated API code by additional semantic conjunctions. This structural pattern is a valuable alternative to sub-classing, allowing the attachment of additional members and methods. This part of the generator, the *decorator API*, serves as the interface between the API code of the eJSL language (model API) and platform-specific code templates of the generator as part of the generator front-end.

Figure 6.1 presents the proposed architectural generator frontend concept including the generated model API which is extended by the decorator API. It consists of interfaces and helper classes which are derived by the original model API. These can be mapped to the platform-specific back-end part of the generator, independent to the targeted platform. The intention behind this decision is to increase the reusability of the generator with template variants as much as possible. So, further refinements of the language and augmentations of the generator can be supported (e.g. to support a new platform version or another system).

¹Design patterns describe reusable architecture guidelines for software systems. For further reading see [74] and [77].

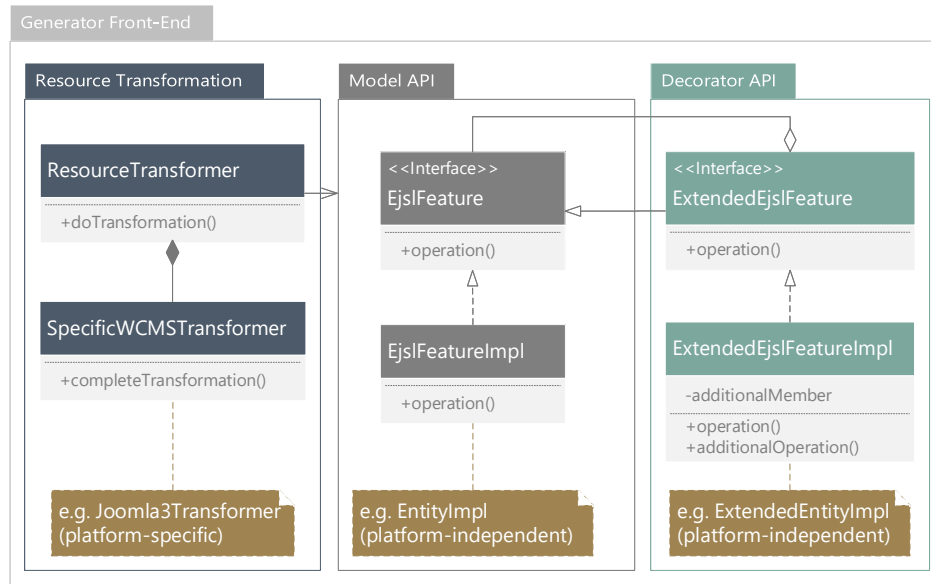


Figure 6.1: Generator Front-End including the Decorator API and Resource Transformer

In order to follow the *convention over configuration* design paradigm, the generator allows abstract models as well as concrete models (smart/dummy approach [66, 193]). Therefore, in order to achieve full extension code, input models become preprocessed in an initial step of the code generation process. To this end, the generator includes a *resource transformer*, which decorates the input model by default model information, required for the code generation process. The resource transformer is a composition of platform-specific transformers which add model elements that are required for a specific WCMS (cf. Figure 6.1). Examples for such elements are entity attributes for WCMS-specific metadata (e.g. IDs, timestamps, or states), standard links from index to details pages, or WCMS-specific extension parameters or languages. To access model elements properly, the generated model API is used. So, the actual model elements can be checked and, if necessary, updated. By implementing transformation logic within the pre-processing front-end part of the generator, the back-end part can stay as lean as possible and the required effort to handle the diversity of language features can be reduced to a minimum. However, the transformer may apply platform-specific transformation rules for a specific WCMS.

Generator Back-End

In the platform-specific back-end part of the generator, we inherently propose an architecture which is close to the structure of the generated code. This allows developers with technical knowledge to augment the platform-specific part of the generator, if needed. Xtend, for instance, provides mechanisms for reusing generator templates by separating them into reusable units.

As Figure 6.2 shows, the architecture of the proposed generator back-end follows a *builder pattern* on two levels. First, we propose to split up the logic for the actual code generation separated by the main features of the DSL: *Entity*, *Page*, and *Extension*. This allows a further extension of the generator, if new features will be added to the language. On the next level, we follow the same pattern for the actual parts of a specific model feature based on the targeted platform. An exemplary implementation is the `ExtensionGenerator` which contains a handler (director) for extension generation for a specific WCMS such as Joomla 3. This handler organizes the specialized code generation for each technological feature of the chosen WCMS (version) by generator templates. Examples for such special features are the variants of extension kinds

which are provided by the WCMSs. This architecture simplifies the augmentation and increases the interchangeability of the generator with templates for other WCMSs. *If another system will be supported by extension generation, a new handler for the new system must be implemented and added to the specific feature generator.* As we present below, we took advantage of this pattern during the implementation of Joomla-specific generator templates (see Section 6.2.2). We implement platform-specific handlers for different major versions of Joomla, which differ tremendously in their code architecture.

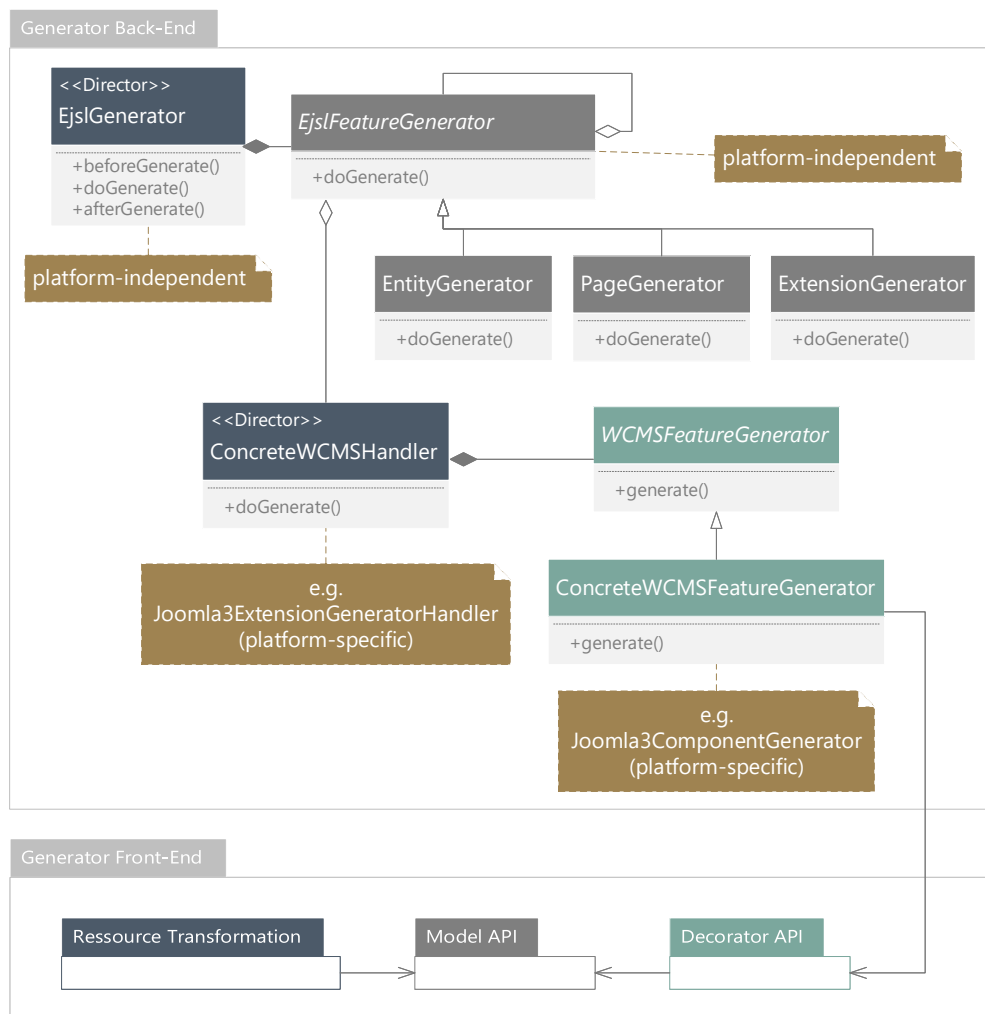


Figure 6.2: Generator Back-End including the WCMS-specific Code Templates

With the decision to split up the back-end part of the generator into a DSL-based and a technology-based part allows full control of the code template separation. The main advantage of this separation is the possibility of using and extending the code templates for both full and partial code generation. If required, only fragments of the generator templates can be used for generating parts of an extension, e.g. if developers want to partially generate the entity-specific (data management) part of an extension. This is achieved by the direction of the **EjslGenerator** class, which contains all specific feature generators. The same applies to the technology-based part, e.g. if only the frontend part of a Joomla component has to be generated.

Particularly worth mentioning is the containment relationship between actual implementations of the `EjslFeatureGenerator` class. This allows us to reuse them for various contexts. The architecture is comparable to the *frame processor architecture* as described in [221]. The code templates are implemented as reusable frames which return variable code templates, based on given arguments, to the actual code generator class. In concrete terms, we use this architecture to allow a full generation of extensions using the *entity* and *page* implementation, whereas the latter is also used for partial code generation. The benefit of this approach is, that the same template code can be reused for various needs without implementing redundant template code.

General Generator Workflow

The whole generation process becomes initiated by the `EjslGenerator` class, which runs the resource transformation and creates instances of the implemented generator classes. The general generator workflow consists of the following steps taking an instance model of the eJSL language as input and creating WCMS-specific extension code:

1. *Preprocessing*: During the initial preprocessing step, the input model becomes decorated by default model information which is required for the subsequent generation process. This task is done by the *resource transformer* (cf. Figure 6.1).
2. *Extension Generation*: In the next step, the generation of platform-specific extension code is performed. To this end, the platform-independent extension generator is initialized. This generator object creates instances of extension handlers, such as respective extension handlers for Joomla 3 or Joomla 4, and invokes the actual platform-specific code generation process. The handlers manage the code generation by invoking feature generator implementations. These contain code templates for the features of the targeted platform, such as the code generation of component manifest files for Joomla 3. For accessing instance model information, the decorator API is used (cf. Figure 6.2). The extension generator also initializes the page and entity generators which create instances of platform-specific page and entity handlers. These, in turn, invoke the generation process of platform-specific extension code based on page and entity definitions in the input model. Examples are list and details views (pages) or SQL installation files (entities) of a Joomla 3 component.
3. *Partial Code Generation (optional)*: In addition to the extension generation, the generator concept also allows the optional generation of partial extension code based eJSL features - so far pages and entities. To this end, the `EjslGenerator` class may instantiate the page or entity generators directly (cf. Figure 6.2) which instantiate handlers for the platform-specific code generation based on the respective feature (see step 2 above). We propose to generate placeholders for all code fragments which require extension-specific model-information which is not given during partial code generation.

6.2.2 Joomla-specific Extension Generator

In this section, we present a prototypical implementation of the previously described generator concept. We decided to use the Joomla WCMS as target platform, since it provides the most sophisticated extension mechanism (cf. Section 2.2.2). For the sake of conducting research on the profitability of MDE in the WCMS domain and addressing the related problem statements in the domain (cf. Chapter 1), we realised platform-specific template variants for two different major Joomla versions (3 and 4) which differ exceedingly in their actual code architecture. So, we can cover the migration scenario, as stressed in this work (scenario 3) and provide a viable solution to reduce required effort during development and migration of Joomla extensions (cf. problem statements 1 and 2). The realised generator can serve as adequate example for further generator implementations for other WCMSs with similar or less extension features. During the development of the generator templates, we followed the described procedure of Section 3.3.6.

Reference Extensions for Joomla 3 and 4

We inspected a variety of reference extensions including Joomla core and popular third-party extensions as well as reference extensions of experienced Joomla developers. In order to fulfil the required extension architecture and to ensure correct extension code as investigated in the first part of this work, the reference extensions have been evaluated based on the community guidelines and, if necessary, re-engineered. The reference extensions were reverse engineered in order to identify generic and schematically recurring code fragments. The selected reference extensions consist of extension features which are important in the context of this work. This includes CRUD-intensive views, internationalisation and localisation features, and interactions with other extensions. Based on the selection of Joomla as target system, we investigated a variety of adequate extension kinds such as components, modules, plugins, templates, and libraries. Extensions with less or no data management are excluded during the investigation, since they are not addressed by our MDE approach. The same applies to non-interoperable extensions which are typically used for very special tasks such as components for backing up a Joomla instance (e.g. Akeeba Backup [3]).

To support the migration scenario with our implementation, we researched Joomla 3 and 4 extensions which differ in their architecture on file and code base. Especially The extension architecture of components differs tremendously between both versions. In contrast to the required MVC structure on file and code base in Joomla 3 (cf. [168]) components, Joomla 4 components must follow an improved file and code structure, still following the MVC approach (cf. [169]). Moreover, Joomla 4 enables an *orthogonal component structure* which separates components into vertical and horizontal [44, 172, 100]. Independent extensions such as the Joomla 4 core extension are developed as *vertical components* (cf. scenario 1), whereas third-party developers can create *horizontal components* to add additional features on top of them (cf. scenario 2). The latter are useless without the vertical components on which they rely on. In addition, Joomla 4 requires extension developers to make use of latest technologies like Bootstrap 4 [173], removed obsolete technologies like the JavaScript framework MooTools [191], and specified PHP 7 as minimum version, supported by the Joomla core. All these innovations must be taken into account when developing suitable generator templates for Joomla 3 and 4 extensions.

Whereas Joomla 3 is in a stable version for years, including matured documentation, as of 2020 Joomla 4 is still in a beta phase. The presented generator in this chapter is based on the Joomla 4 alpha release from 2019 (Alpha 12 [167]), following the architecture which was documented during this time. A stable release is planned for late 2020. In the following sections, we present the differences between the two Joomla versions including an illustration of the required file structures for each version. However, we will not present differences on code level, due to the constantly changing code base. For further reading see the developer documentation for Joomla 3 [168] and Joomla 4 [169].

Preprocessing

As previously described, instance models of the eJSL language undergo an automatic preprocessing step. So, abstract instance models become annotated before they are used during the actual code generation process. The preprocessing step is executed by an implemented resource transformer which includes platform independent and platform-specific decorations for model instances.

Considering the *data modelling (entities)* part of an eJSL instance model, several preprocessing operations are executed. Since Joomla is based on a relational database, *primary attributes* are required in order to allow relationships along relations. Therefore, each entity is decorated by a primary attribute, if none is defined. The DSL allows to create references between entities without specifying primary attributes. To keep the generator as lean as possible, we decided to extend

existing references by the newly created primary attributes, if necessary. This allows more flexible generator template logic, since both attributes are part of the reference, the original attribute and, if automatically added, the new primary attribute. A more complex preprocessing step is executed, if a *many-to-many cardinality* between two entities is modelled. In order to support the code generation for the relational database, these relationships are resolved by the creation of a new entity with relationships to the original entities (cf. Figure 6.3). This entity serves as linking entity which can be used for the generation of an associative table for the relational database. The original attributes are preserved in the original entities in order to remember the original reference attributes. These may be valuable during further code generation. In addition to these actions, Joomla-specific standard attributes like **state**, **ordering**, and **parameters** are added to all specified entities.

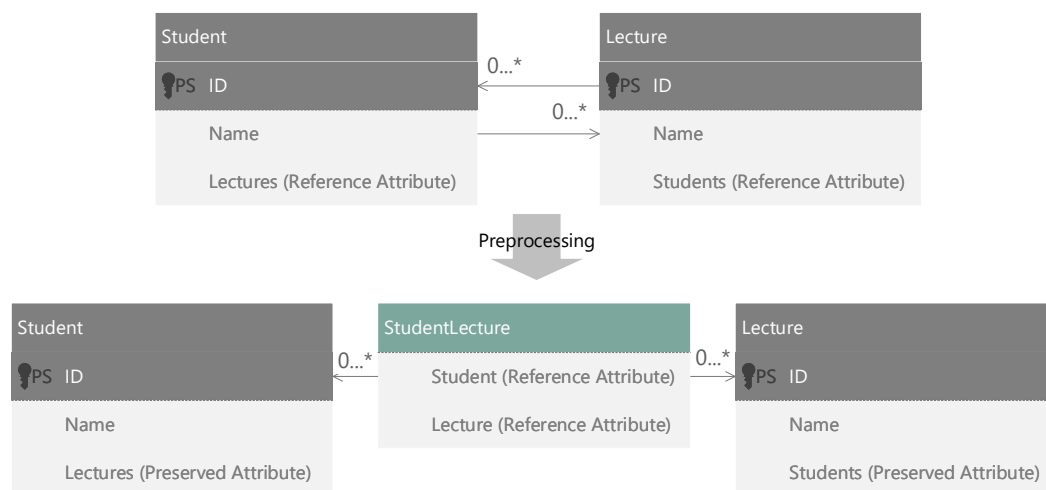


Figure 6.3: Preprocessing of many-to-many Relationships between two Entities

The *interaction modelling (pages)* part of an instance model is refined in order to complete index and details page definitions. So, if not specified, each index page definition becomes decorated by **representation columns** (table columns) and **filters** based on the attributes of the referenced entity (cf. Figure 5.8 on Page 96). By default, we decided to add all entity attributes, except automatically added primary key attributes, as representation columns and filters. Additionally, a default context parameter in a link definition is added. This parameter is selected based on the primary attribute of the referenced entity. Details page definitions are extended by **editFields** specifications based on the respective attribute type of the referenced entity. The type mapping is shown in Table 6.2.

Table 6.2: Type Mappings for eJSL Standard Types and HTML Types

Attribute Type	HTML Type
Integer	Integer
Decimal/Currency/Short_Text	Text_Field
Boolean	Yes_No_Buttons
Text	Textarea
Time/Date/Datetime	Datepicker
Link	Link
Image	Imagepicker
File	Filepicker

The following listings illustrate the page part of an eJSL instance model before (Listing 6.1) and after (Listing 6.2) the preprocessing step.

```

1  IndexPage Participants {
2      *entities = Participant
3      links {
4          InternalContextLink Details {
5              target = Participant
6              linkedAttribute = Participant.name
7          }}
8
9  DetailsPage Participant {
10     *entities = Participant
11     links {
12         InternalLink Index {
13             target = Participants
14             linkedAttribute = name
15         }}

```

Listing 6.1: Example of Page Defintion before Preprocessing Step

```

1  IndexPage Participants {
2      *entities = Participant
3      representationColumns = Participant.name, Participant.affiliation ,
4      filters = Participant.name, Participant.affiliation , Participant.
5      links {
6          InternalContextLink Details {
7              target = Participant
8              linkedAttribute = Participant.name
9              linkParameters {
10                 Parameter id = *Attribute "Participant.id"
11             }}}
12
13  DetailsPage Participant {
14      *entities = Participant
15      editFields {
16          Participant.name {
17              htmlType = Text_Field
18          },
19          Participant.address {
20              htmlType = Textarea
21          },
22          Participant.affiliation {
23              htmlType = Textarea
24          },
25          Participant.nationality {
26              htmlType = Textarea
27          }
28      }
29      links {...}
30  }

```

Listing 6.2: Example of Page Defintion after Preprocessing Step

With regard to the *extension modelling* part, two aspects are focused by the resource transformer. First of all, the *metadata* part (**manifest**) of an instance model is extended, if not all language features are specified. This includes extension information such as copyright, license, version, and the creation date. The second decoration considers the *internationalisation and localisation* of the extension which is specified in the **languages** part. If no extension language is specified, **en-GB** (English/Great Britain) is added. Moreover, the language constants for all extension-specific text labels are added. So, this logic can be maintained within the preprocessing part of the generator and must not be part of the actual code generator templates.

Output Description and Generation Logic

As previously described, most of the actual MDE infrastructure implementation is realised with Xtext and Xtend. We use Xtend for template realisations, whereas Xtext is used for the realisation of the eJSL DSL (cf. Chapter 5). By using these tools together, we ensure rapid development of a high quality MDE infrastructure under consideration of the above-mentioned requirements (cf. Section 4.3). The implemented generator uses eJSL model instances as input and supports the generation of full extension code for two major Joomla versions (3 and 4). To this end, the platform-specific parts of the generator were implemented for both Joomla versions allowing developers to choose between the different versions. So, developers can flexibly generate Joomla extensions for their required Joomla version based on the same eJSL instance model. The generator provides *model-to-code transformation* for common extension types which are supported by the Joomla WCMS. However, we mainly focus on components and modules, due to their sophisticated functionality and interoperability in comparison to other extension kinds. During the development, we started with the development of generator templates for version 3 of the Joomla platform. Then, we made use of the proposed generator architecture concept in order to implement generator templates for the latest major version 4. As of the time of implementing the templates in 2018 and 2019, the latest platform version was in an alpha state (Alpha 12 [167]). Based on the current development of the project, similar extendibility features are provided. So, the platform-independent parts of the generator were directly reusable or could be copied. The same applies to the platform-specific parts of the most simple extension kinds (plugins, libraries, and templates).

As stated in [114], descriptions of actual code generator implementations are typically made by an adequate *output description* for the desired domain and the implemented *generation logic*. The output description addresses the architecture of the generated artefacts, whereas the generation logic describes how the input is mapped to the output. We consider both aspects in the following presentation of the Joomla-specific generator implementation. This includes the architectural variety of and dependencies between generated Joomla extensions. For a better understanding, the already known conference model is used as running example. This running example is simple enough to present the provided features of the generated code and is close to a real-world extension project. As previously mentioned, we discuss differences between the two stressed Joomla versions, such as differing file structures, without presenting code output due to the constantly changing code base.

Supported Extension Kinds and Extension Metadata: All supported extension kinds of the Joomla WCMS are covered by the presented generator implementation. However, components contain the most sophisticated architecture following a model view controller pattern on code and file base. Additionally, the variation between a component's frontend and backend view becomes considered as well, what results in suitable code for the respective section. Moreover, components may include their own data management by an extension of the underlying database of the hosting Joomla instance. Therefore, the development of adequate generator templates for this extension kind was the most tedious process. Especially, the essential differences between Joomla 3 and Joomla 4 components required the most effort during the evaluation of reference components and iterative generator development. The main feature of the generated code is its architectural

structure based on the specific extension kind. For all extension kinds, the required manifest and language files are generated in the common Joomla structure. So, generated extensions can be installed properly to a host instance. *Manifest files* contain all required extension information under consideration of the specific extension kind (cf. Listing 2.4 on Page 25). The metadata information is inferred from the **Manifest** definition in the extension part of an eJSL model instance. To support the multi language feature of Joomla, language files which consist of key-value pairs are generated, whereas the language keys are also generated into the respective sections of the extension, e.g. the views of a component. Listing 6.3 shows an exemplary language definition within an instance model, whereas Figure 6.4 illustrates the generated Joomla-specific language files and an excerpt of the generated key-value structure. This feature is similar for both supported Joomla versions. Therefore, the generator templates are quite similar.

```

1 languages {
2   Language en-GB
3   {
4     keyValuePairs
5     {
6       SUCCESS_MESSAGE = "This was successful!"
7       ...
8     }
9   }
10  Language system en-GB {}
11  Language de-DE {}
12  Language system de-DE {}
13 }

```

Listing 6.3: Language Specification for an Extension in an eJSL Instance Model

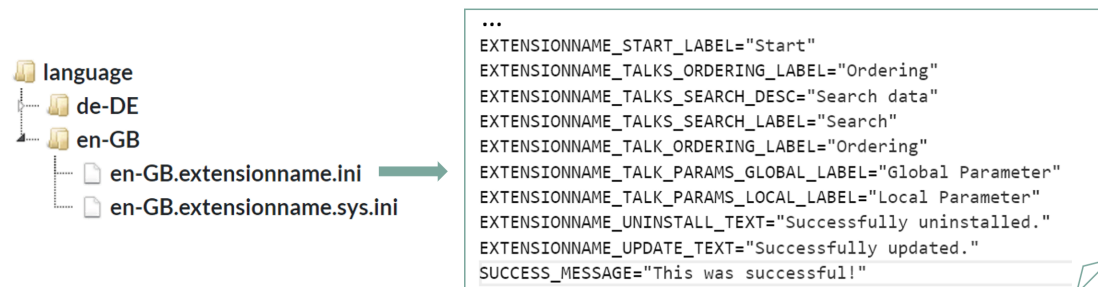


Figure 6.4: Generated Language Files and File Contents

Generation of Joomla Components: Usually, MDE proposes to generate the smallest possible amount of code. However, this is not possible or suitable for applications like Joomla components, since they have to follow a certain structure to ensure a homogeneous behaviour in combination with the hosting platform. Hence, the components which are generated by the presented code generator consist of the same amount of generic and schematically redundant code as if they were developed in the conventional way by hand. However, the generator does not generate cryptic identifiers but rather creates code which cannot easily be distinguished from handwritten code. This applies for the generator templates addressing both supported Joomla platform versions.

If a modelled component contains page references within its backend section the generator creates Joomla code following the standard MVC architecture (cf. [112]). This architecture is extensively described for Joomla 3 components in [109]. A description of the new MVC architecture of Joomla 4 can be found in [108]. Figure 6.5 illustrates the supported MVC architecture of Joomla components.

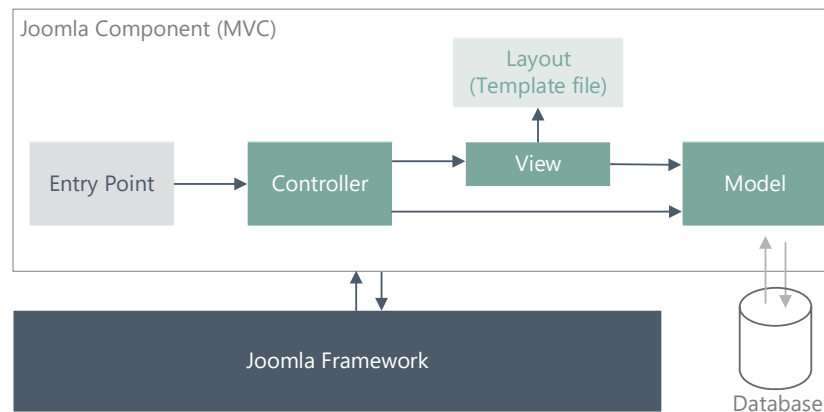


Figure 6.5: Architecture of Joomla Components

An *entry point* PHP file is required for the determination of the correct controller which has to be called. To this end, the entry point file uses a request parameter which includes the action to perform. So, the suitable method of the correct controller instance can be called. The *controller* analyses the request and creates an instance of the necessary model. The *model* isolates the details of data access from controllers and views. It is typically responsible for database requests but can also obtain data from any source. Based on the task, the model is called by the controller in order to update the database, e.g. during a CRUD operation. Another responsibility of the controller is the instantiation of and redirection to the view which is requested by the WCMS user. The *view* knows its corresponding model which is called in order to obtain the data which has to be used for the actual HTTP response. Moreover, the view calls adequate template files (*layouts*) which dynamically specify the actual HTML output for the view.

Entry points of deployed components are called by the Joomla framework based on the request parameters which must include the component. The correct component is assigned to corresponding menu items automatically during the installation of a component for the administration section, or manually by administrators for the frontend section. Figure 6.6 illustrates an automatically created menu entry in the backend section of a Joomla instance. This menu entry is a link to the deployed conference component. To this end, `option=COM_MYCONFERENCE` added as request parameter to the link. So, the entry point of the component is called.

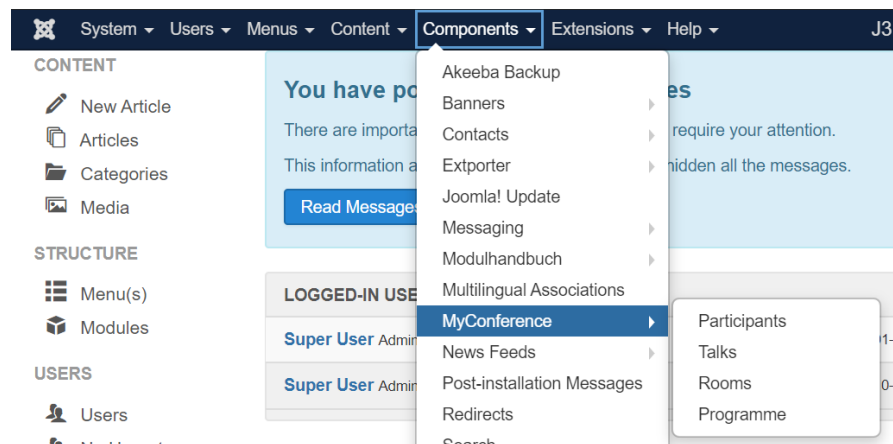


Figure 6.6: Automatically Created Component Link as Menu Item in the Administration Section of a Joomla Instance (J3)

Such interoperability features between the Joomla framework and deployed components necessitates the implementation of the required MVC architecture on code and file base. So, a homogeneous operability of deployed components can be ensured. Even though the same architecture is supported by both considered major platform versions, the file and code structure of components differ tremendously. Therefore, we provide respective generator templates for both versions which adhere to the respective file and code architecture (cf. Figure 6.7).

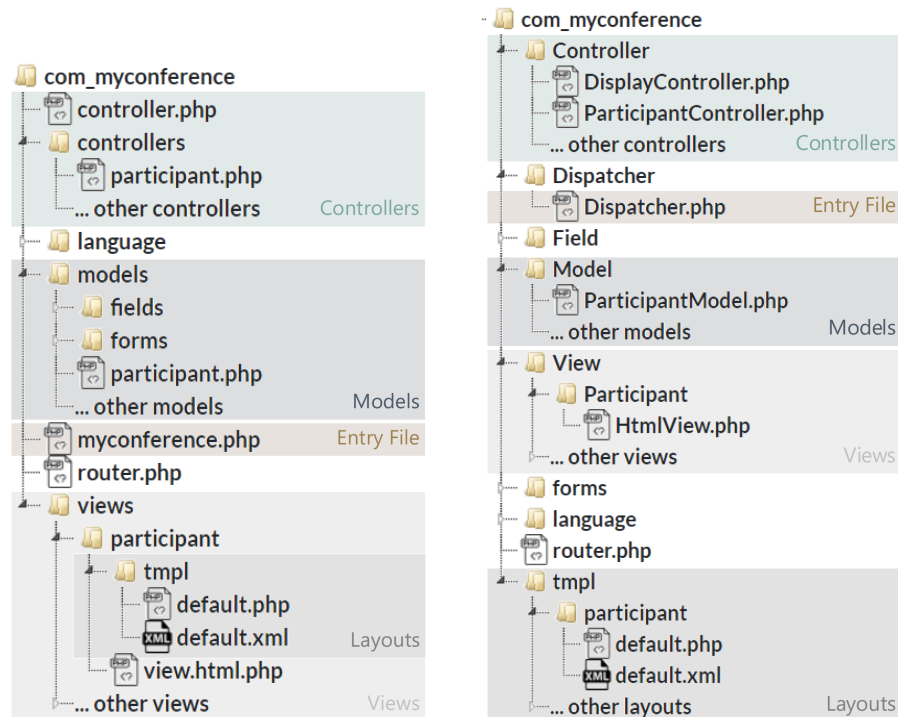


Figure 6.7: Generated File Structure of a Joomla 3 (left) and 4 (right) Component

With the generated component views, users can operate on the entities which are referenced by the pages of the modelled component. This comprises all CRUD operations in consideration of the sophisticated permission management of a Joomla instance, including standard actions like ordering, filtering, and search operations. Once installed, the look and feel of the generated views is similar to other views of handwritten extensions which are installed. So, administrators can operate on data entities managed by the component in the common way as Figure 5.3 on page 91 illustrates. Since there is no standard for the frontend section of a component view, we decided to follow a similar view design. However, in the frontend section the permissions are considered differently. Therefore, the generator also creates a details view which presents the details of an entity without the possibility for edit operations. This view is generated automatically in addition to the edit details view, if a details page is referenced by the frontend section of a modelled component. If the frontend user has the permission to edit the entity item, additional buttons are illustrated which open the edit view or allow to delete an entity (see Figure 6.8).

Typically, frontend views can be assigned to menu items. This is considered by the code generator as well. However, the generated details views, which are not intended to be assigned to a menu item are excluded from the assignable views. Such views typically require context information such as the id of the entity item. In order to provide CRUD operations on component entities also for frontend users, the generator creates similar business logic for the frontend. This feature is not common for usual Joomla components, but was required by the interviewed developers and users from the community.

state	✓
Created By	Super User
Name	Dennis
Affiliation	THM
Nationality	DE
Address	Wiesenstraße
<input type="button" value="Edit"/> <input type="button" value="Delete"/>	

Figure 6.8: Frontend Details View of a generated Component

Handling references between entities is one of the most complex parts during code generation of Joomla components. We decided to generate different HTML constructs in dependency of the upper limit for the specified reference. Our proposed generator is able to generate code for references with an upper bound of 1 and -1 (unlimited) independent of the minimum limit. Other multiplicities and the definition of -1 as upper bound within an unidirectional reference are currently not supported by the generator. Therefore, no logic for managing collections of referenced attributes is generated. However, it is planned to include this feature into a further version of the generator. Unidirectional reference specifications of 1, such as illustrated in Listing 6.4, are translated to a suitable form field (Select) in edit views of referencing entities (cf. Figure 6.9). The instance model includes a reference from the **Session** entity to the **roomname** attribute of the **Room** entity. So, the generator creates a select form field which includes already created rooms as options in the edit view of sessions. This view is used during the creation and editing of sessions. This feature is generated without the need of explicit specifications in the pages part of the model which includes the page definitions.

```

1 Entity Room {
2   attributes {
3     Attribute roomname {...}
4     Attribute position {...}
5   }
6
7 Entity Session {
8   attributes {...}
9   references {
10    Reference {
11      entityAttribute = room
12      referencedEntity = Room
13      referencedEntityAttribute = Room.roomname
14      max = 1
15    }
16  }
17 }
```

Listing 6.4: Entity Definition with unidirectional Reference

Bidirectional references without upper limits (many-to-many) are considered implicitly during the generation of both list and details views. The example in Listing 6.5 illustrates such a relationship by two entities with references to each other. The translated details views are augmented by a new tab for the assignment of already defined entities that can be referenced (cf. Figure 6.10). The new tab includes a Multiselect form field which allows to append existing instances of the targeted entity. Moreover, the generator also considers multiple references in

Figure 6.9: Form Field for Referenced Attributes

the list view (cf. Figure 6.11). All assigned references are shown in the table column, if it is specified as **representation column** in a page definition within the input instance model. If the referenced entity attribute is used within a link specification in the list page definition, links to e.g. the details view of the reference are generated. The implemented generator also allows multiple bidirectional references between two entities. In such a case, several tabs are generated for the corresponding details view, one for each reference.

```

1 Entity Participant {
2   attributes {...}
3   references {
4     Reference {
5       entityAttribute = talks
6       referencedEntity = Talk
7       referencedEntityAttribute = Talk.title
8       max = -1
9     }
10  }
11 Entity Talk {
12   attributes {...}
13   references {
14     Reference {
15       entityAttribute = speakers
16       referencedEntity = Participant
17       referencedEntityAttribute = Participant.name
18       max = -1
19     }
20  }

```

Listing 6.5: Entity Definitions with bidirectional Reference (Participant \longleftrightarrow Talk)

The generated output of references between entities is analogously implemented for both addressed Joomla versions, whereas the generator templates adhere to the platform-specific file and code structures. The generator creates no custom styles for any form element. So, the appearance of the views only depends on the activated template of the Joomla instance.

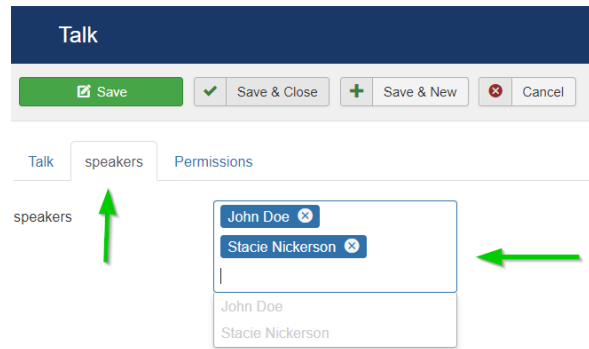


Figure 6.10: Additional Tab in Edit View with Form Field for Multiple References

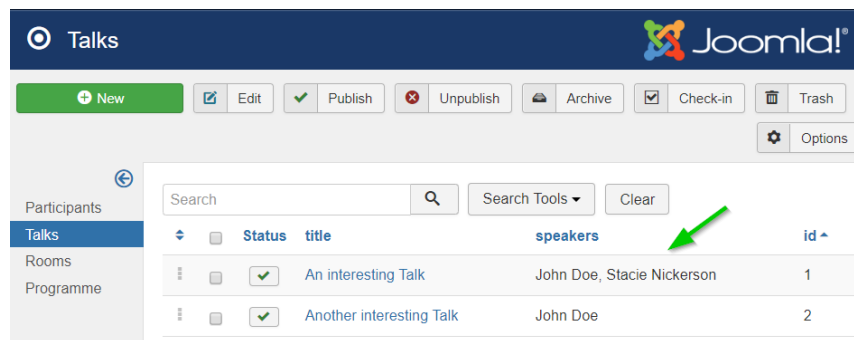


Figure 6.11: Representation of Multiple References in a List View

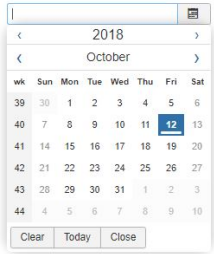
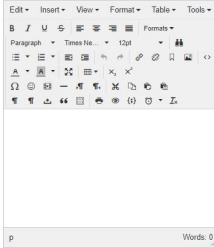
As stated before, the eJSL grammar provides a set of abstract *data types* (**attribute types**) which can be used for entity attributes. These types become translated to suitable SQL types for the database script during code generation of components (see Table 6.3). These types correspond to standard types which are used by Joomla and third-party extensions. For the representation of entity attributes in page implementations, *HTML types* are provided. So, modellers can decide how the attributes will be represented in e.g. an edit view of a component. If, for example, an entity attribute has the abstract **Short_Text** attribute type in an eJSL instance model, the type gets mapped to **varchar(255)** in the SQL file.

Table 6.3: Type Mappings for eJSL Standard Types and SQL Types

Attribute Type	SQL Type
Integer	INT(11)
Decimal	DECIMAL(10,10)
Currency	DECIMAL(13,4)
Boolean	TINYINT(1)
Text/ Link/ Image/ File	TEXT
Short_Text/ Time	VARCHAR(255)
Date	DATE
Datetime	DATETIME

If no specific HTML type is defined within a page definition, the resource transformer creates a suitable type based on the mapping shown in Table 6.2. All attribute types which are part of a page in the given instance model are mapped to HTML types (form fields). With this information the generator creates suitable HTML implementations which can be found in Table 6.4.

Table 6.4: Type Mappings for eJSL HTML Types

HTML Type	Supported Attribute Types	HTML
Integer	Integer, Decimal, Currency	<input type="text"/>
Yes_No_Buttons	Boolean	<input type="radio"/> Yes <input checked="" type="radio"/> No
Textarea	Text, Short_Text, Integer, Decimal, Currency, Time, Date, Datetime, Link, Image, File	<input type="text"/>
Text_Field	Text, Short_Text, Integer, Decimal, Currency, Time, Date, Datetime, Link, Image, File	<input type="text"/>
Link	Text, Short_Text, Link	<input type="text"/>
Datepicker	Time, Date, Datetime, Text, Short_Text	
Imagepicker	Image, Text, Short_Text	<input type="button" value="Upload Image"/>
Filepicker	File, Text, Short_Text	<input type="button" value="Upload File"/>
Editor	Text, Short_Text	
Select	Integer, Text, Short_Text	<input type="text" value="male"/>
Checkbox	Text, Short_Text, Integer, Decimal, Currency	<input checked="" type="checkbox"/> option1 <input checked="" type="checkbox"/> option2 <input type="checkbox"/> option3
Radiobutton	Text, Short_Text, Integer, Decimal, Currency	<input type="radio"/> option1 <input checked="" type="radio"/> option2 <input type="radio"/> option3
Multiselect/Encrypted_Text/Hidden	Text, Short_Text	n.a.

The table lists the supported attribute types and shows the generated HTML code as interpreted by the browser. Whereas form fields for HTML outputs of forms like edit views can be explicitly specified in the layout files, Joomla provides a more elegant way for building the HTML output. The Joomla framework allows the description of forms within a descriptor file (XML). These forms consist of definitions for each form field (cf. Listing 6.6). Each declared form field is translated automatically to the suitable HTML output by the Joomla framework. Moreover, Joomla allows to specify custom field types which can be referred in form definitions. Based on the supported HTML types of the eJSL DSL, form files for required pages are generated. The same applies to custom form fields, which are used for references between entities, as described above.

```

1 <field name="affiliation"
2   type="editor"
3   id="affiliation"
4   label="COM_MYCONF_FORM_LBL_PARTICIPANT_AFFILIATION_LABEL"
5   description="COM_MYCONF_FORM_LBL_PARTICIPANT_AFFILIATION_DESC"
6 />

```

Listing 6.6: Field Definition in A Form Specification File

Even though the HTML types can be mapped to various standard attribute types, such as a HTML textarea which can represent an attribute of type **Text**, **Image**, **Date**, and more, the generator does not check their consistency, since this task is done during modelling by model validators.

Generation of Joomla Modules: In contrast to components, modules are a special extension kind, since they usually depend on existing data, e.g. from components. Therefore, if a component is referred to a module, dependencies to components are generated in the module to ensure a proper interplay within the host instance where both component and module are installed. So, if everything is modelled correctly, modules can be installed and used without any handwritten code. Figure 6.12 illustrates the common architecture of Joomla modules.

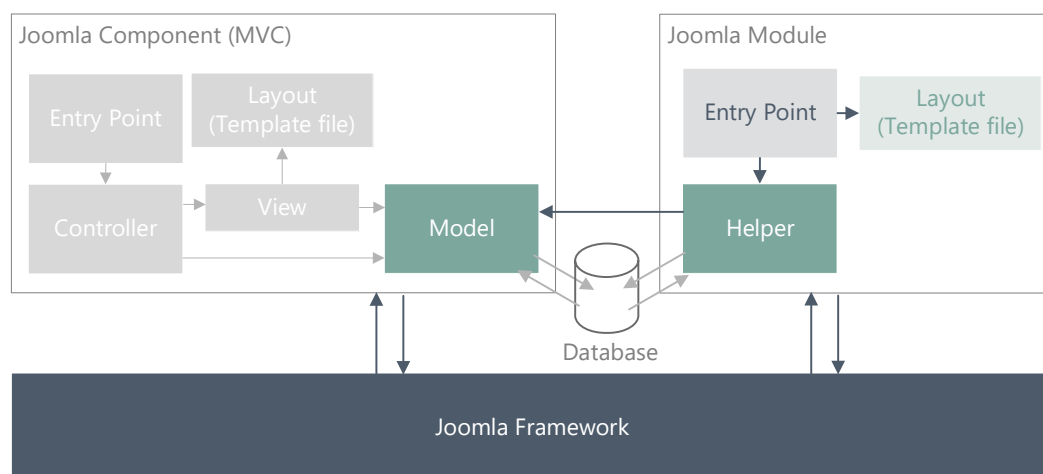


Figure 6.12: Architecture of Joomla Modules

Even though this architecture is not strictly required by the Joomla framework, most of the core and third-party modules follow this file and code pattern. Similar to components, modules consist of an entry point. This entry point is called by the Joomla framework of the hosting Joomla instance, when users request a page which includes one or more instances of the module.

If a module entry point is called, it returns the layout file which is used as part of the response for the requested page. These layout files are similar to the layouts which are used by component views. As already mentioned, modules typically illustrate data from any existing data source. This aspect is covered by the eJSL DSL. The language allows to assign a data access object (DAO) of a referenced component (**frontendDAO** or **backendDAO**) as data source. Additionally the **database** of a component or a **webservice** can be specified as data source. Though, the most common data source of a module is a corresponding component which manages the accessed data. Therefore, a common implementation is the re-use of respective component models as reference within the helper file of the module. So, the model can be used within the layout file in order to obtain data via the component model. Though, this requires that the component is also deployed to the same hosting Joomla instance as the module. Such dependencies can be defined in the model by referencing a page which is also used within a corresponding component and the assignment of **backendDAO** as data source (see Listing 6.7). The assignment of **frontendDAO** leads to similar code using the frontend model of the specified component. Alternatively, code for direct database access for component-specific database tables can be generated. In the case that no dependency is specified, our generator creates placeholders for the required identifiers of the preferred data source. This, however, requires handwritten customizations after code generation. The provided DSL feature for using a web service as data source is currently not covered by the presented generator, due to the lack of adequate reference extensions.

```

1 Module Talks {
2   Manifest {...}
3   languages {...}
4   *Page : Talks from : MyConference data backendDAO
5 }

```

Listing 6.7: Module Dependency to a Model of a Component

The generated module files and code correspond to the common module architecture based on the most core and third-party extensions which were evaluated. In addition to the manifest and language files, our generator creates an entry point file, a layout which defines the HTML output, and a helper file. The latter is used for data access and depends on the selected data source in the instance model. The structure of generated Joomla 3 and 4 modules is quite similar as the example in Figure 6.13 shows.

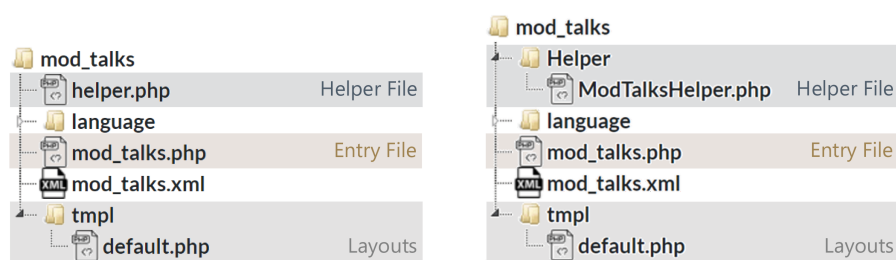


Figure 6.13: Generated File Structure of a Joomla 3 (left) and 4 (right) Module

We decided to use given model information of page specifications in order to enhance common module features during module configuration in the administration section of a Joomla instance. So, in addition to common configuration features for menu assignment and permission specification, our generated modules allow administrators to define which data is shown by the module. So, in dependency of the selected page kind of the referenced page in the eJSL instance model, one specific or a list of all entity instances is shown (see Figure 6.14). The latter, can be limited, ordered, or filtered by entity attributes of existing entity instances. To this end, we use the same logic as in components (cf. Figure 6.15).

You are here: [Home](#) > [Programme](#)

Room	Talk	Time	Title
123	Design Patterns	2016-04-16 10:00:00	Design Patterns
14	JooMDD - Simple Joomla Extension Creation	2016-04-16 14:00:00	JooMDD
25	Internet of Things	2016-04-17 11:00:00	IoT

Conf_talks

- JooMDD - Simple Joomla Extension Creation Talk about JooMDD Dennis Priefer
- Design Patterns John Doe
- Internet of Things Yaddow Green

Figure 6.14: Representation of a Generated Module illustrating the Data of an installed Component in the Frontend of a Joomla Instance)

Modules: mod_talks

Joomla!

Title *

Conf_talks

Module

Menu Assignment

Filter: Advanced Permissions

Created By

title

Select

speaker

Select

id

Select

Dennis

John Doe

Figure 6.15: Configuration of a Generated Module in the Administration Section

Our implemented generator does not support any kind of UI modelling. Therefore, generated modules offer an unattractive representation. We suggest a manual refinement by adding custom style sheets with adequate selectors for the generated HTML elements in the module.

Generation of Joomla Plugins: In addition to components and modules, the presented Joomla generator also supports the translation to Joomla plugins. If a plugin is generated, the code generator considers the selected plugin kind in the model. So, the specific plugin class of the Joomla core can be extended. As stated before, plugins are not dependent of a specific section of a Joomla instance. They provide actions which are performed, if specific events are triggered by the hosting system. Such events occur, when a user clicks the search button, after login attempts, or before an article is loaded. In [113], further information about Joomla plugins can be found.

According to the coding guidelines of the Joomla community, Joomla plugins must consist of at least one PHP class file in addition to a manifest and supported language files. The plugin class must extend a plugin class of the Joomla framework in order to ensure a correct operability on a hosting Joomla instance. All deployed and activated plugins are then automatically instantiated by the Joomla framework and the appropriate methods based on the event that occurred are called. Based on the plugin type, plugins may access existing data sources such as the database, e.g. for content preparation, user authentication, or the gathering of context-specific search results (cf. Figure 6.16). Our presented generator adheres to given plugin standards and, based on the selected plugin type in the input DSL instance, a class with suitable methods is generated.

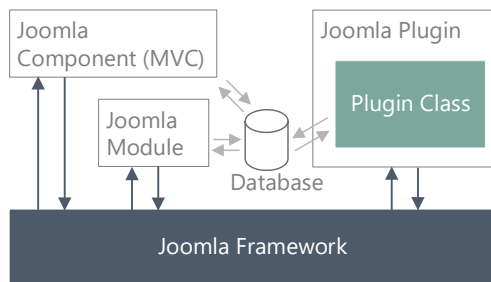


Figure 6.16: Architecture of Joomla Plugins

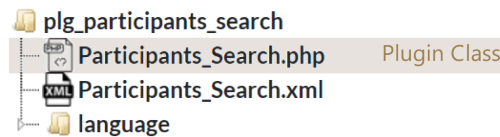


Figure 6.17: Generated File Structure of a Joomla 3 and 4 Plugin

Listing 6.8 shows an excerpt of a generated class of a search plugin. Since the DSL does not support an explicit definition of plugin actions, placeholders for custom code are generated.

```

1 class PlgSearchParticipants_Search extends CMSPlugin
2 {
3     public function plgSearchParticipants(&$subject , $params)
4     { ... }
5     ...
6     public function onContentSearch($text , $phrase = '' , $ordering = '' ,
7         $areas = null)
8     {
9         ...
10        $searchText = $text; //not always used
11        ...
12        switch ($phrase) {
13            case 'exact':
14                //TODO: place code here
15                break;
16            case 'all':
17            case 'any':
18            default:
19                //TODO: place code here
20                break;
21        }
22        ...
23        $query = $db->getQuery(true);
24        $query->select('/*some_code_here*/')
25            ->from('#__Participants AS a')
26            ->where('/*some_code_here*/')
27            ->order($order);
28        $db->setQuery($query, 0, $limit);
29        $rows = $db->loadObjectList();
30        if ($rows) {
31            foreach ($rows as $key => $row) {
32                //TODO: place code here
33            }
34        }
35        return $rows;
36    }

```

Listing 6.8: Generated Search Plugin Class (Excerpt)

Generation of Joomla Templates and Libraries: The presented generator implementation also creates installable template extensions. However, they will not be explained in detail, since they mainly consist of HTML definitions for template positions within PHP files and CSS definitions which are part of the template assets (cf. Figure 6.18). The same applies to generated libraries since they mainly consist of PHP classes, representing the class hierarchy as specified within an eJSL model (cf. Figure 6.18). The benefit of modelling and generating these two extension kinds seems rather low in contrast to manual development.

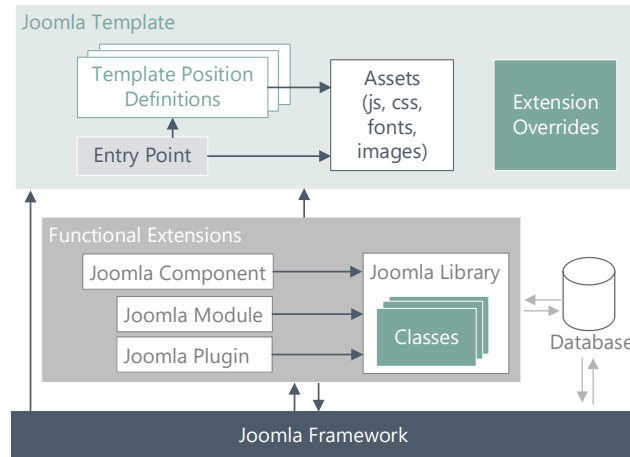


Figure 6.18: Architecture of Joomla Templates and Libraries

However, for reverse engineering purposes, e.g. for a model-driven migration, all extension kinds are supported and integrated as complete as possible (cf. Figure 6.19). Even though HTML override definitions in templates are often used to customize the appearance of core or third-party extensions, they are not considered by the eJSL DSL. However, our generator creates at least a placeholder file, to support developers during the creation of overrides. A drawback of the current generator implementation is the lacking Joomla 4 support. Since no adequate reference extensions existed during development of the generator, we decided to generate the same file structure as for the Joomla 3 platform. Based on the current stand of Joomla 4, our generated templates and libraries appear to be operable, due to the backwards compatibility of Joomla 4.

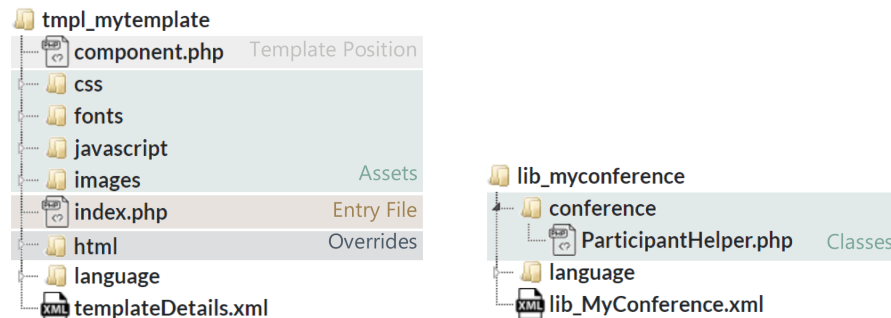


Figure 6.19: Generated File Structure of a Joomla Template and Library

Interactions between Extensions: Besides the specifically tailored code generation, based on the extension kind, one of the most sophisticated features of the presented code generator is the implicit code translation based on interaction definitions between components and modules in an input model instance. As already described in Chapter 5, interdependencies between extensions

can be explicitly defined with the eJSL DSL. These dependencies can be specified on a high abstraction level. The generated code, however, must be able to create correct dependencies to ensure a proper interaction within hosting Joomla instances on which depending extensions are deployed. If modules have an explicit reference to a component, as line 5 in Listing 6.9 illustrates, the respective dependencies will be generated based on the given model information. In the given example, the generated module code consists of a reference to the model file (DAO) of the component `MyConferenceComponent`.

```

1 Module MyConferenceModule {
2   Manifest {...}
3   languages {...}
4   *Page: Participants from: MyConferenceComp data backendDAO
5 }
```

Listing 6.9: Module Dependency to a Page of a Component

However, if no data source assignment or component dependency is placed in the model, the generator has to determine a suitable default implementation. To this end, the generator follows different strategies, if components are placed together with dependent modules within the special **package** extension kind (cf. Listing 6.10). In this case, the generator is able to translate dependencies automatically without explicit references in the model. This, however, requires that the referenced **page** in a module specification is also referenced by a component within the extension package. If dependent extensions are not placed within an extension package, the generator creates placeholders in the code, which must be filled manually after code generation.

```

1 ExtensionPackage MyConferencePackage {
2   ...
3   extensions {
4     Component MyConferenceComp {...}
5     Module MyConferenceModule {...}
6   }
7 }
```

Listing 6.10: Component and Dependent Module as Part of an Extension Package

Extension Updates: Considering iterative refinements of generated extensions, the implemented generator makes use of the extension update mechanism provided by the Joomla WCMS. This allows the re-installation of new extension versions without the need of uninstalling previous versions. During the installation process, existing extension files are replaced by the updated files. If files have to be deleted, script files can be added which are executed automatically during the installation routine. If database changes have to be executed, a higher version number is required in the manifest file. Joomla supports the implementation of database update scripts for explicit extension versions. These are processed even successively, if extension versions have been skipped. Moreover, our generator can also be used for partial code generation. So, only the frontend part or separate views of a component, SQL scripts, or the helper file of a module can be replaced by newer versions.

6.3 Extraction of Deployed WCMS Extensions

As described in Chapter 4, most of the discussed scenarios require definitions of legacy extensions on model level. So, forward engineering of dependent extensions or migration and modernization steps can be applied in a model-driven manner. In order to reverse-engineer the features of an existing extension, it should be available as installable extension package. In this case, all corresponding files can be found in one place. Whereas WCMSs like WordPress or Drupal

maintain installed extensions within a specific extension folder, systems like Joomla uncompress extensions and keep their files separately. As previously described, Joomla components are subdivided during the extension installation routine, whereby the files of the extension package are placed in different folders (cf. Figure 6.20). In the context of Joomla, no tools exist which support developers during the extraction of extension packages from already installed extensions (cf. problem statement 5).

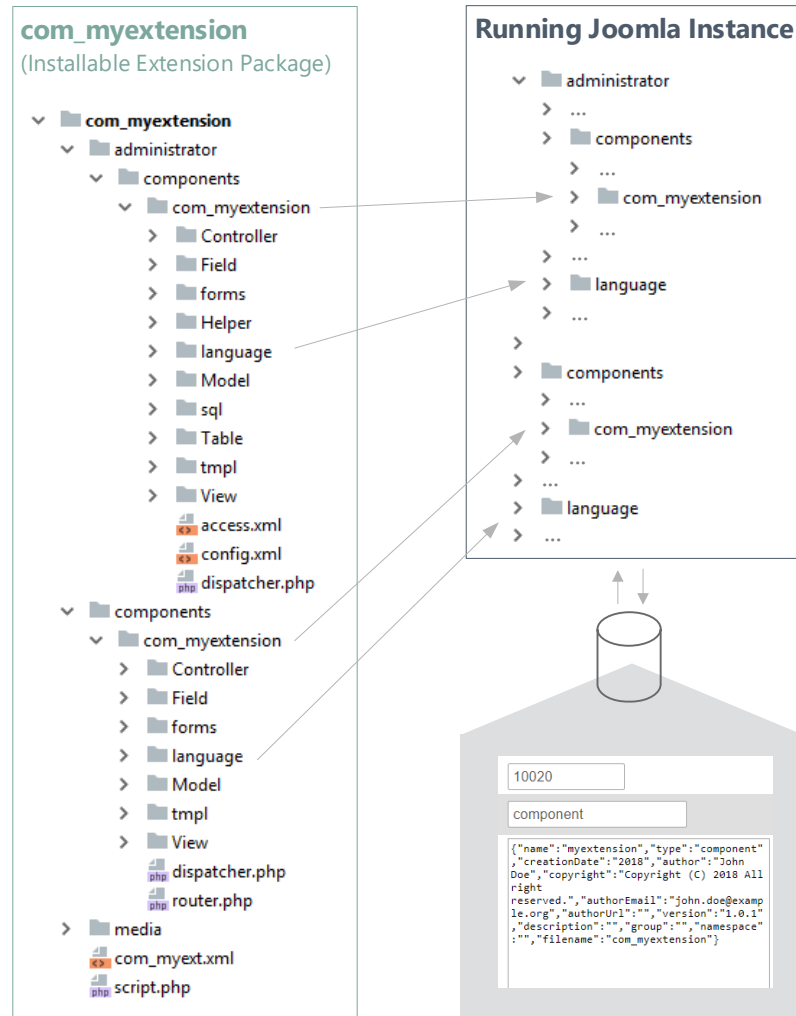


Figure 6.20: Separation of a Deployed Joomla Component

An extraction of extension information from installed Joomla components can therefore be a challenging task (cf. problem statement 3). Fortunately, third-party extensions are typically available as installable extension package, e.g. within extension repositories. However, if this is not the case and an installed extension has to be used during one of our addressed MDE scenarios, developers have to extract it from the running WCMS instance first. The same applies to core extensions, which are typically not available as installable extension package. As described in Requirement 4.3.4, WCMS extension developers require support during the extraction process of a deployed extension, especially for complex extensions as Joomla components. Therefore, *we propose an approach for the automatic extraction of deployed WCMS extensions* from running WCMS instances in this section. We introduce a general concept and present a platform-specific implementation for Joomla components.

6.3.1 Concept

In contrast to a manual extraction of deployed WCMS extensions, we propose an automatic extraction and extension package export provided by tool support. So, developers can be assisted during the application of the addressed development scenarios. Moreover, MDE infrastructure developers can be supported in order to gather extension features from reference extensions for the iterative refinement of adequate code generators. Figure 6.21 illustrates a concept which consists of various actions in order to automatically create installable WCMS extension packages based on deployed extensions. These actions are:

- *Extract deployed extension files.* This includes all corresponding extension files, which are spread over different folders in the filesystem of a WCMS instance.
- *Extract the database scheme* from the database. This allows the creation of adequate database scripts for the extension.
- *Gather concrete extension data* from the database. So, the concept addresses development scenarios which include the migration of deployed extension data.
- *Build an installable extension package* which aggregates all extracted extension files in the correct structure as required by the WCMS. Moreover, the package must include database scripts which contain the database scheme and concrete data of the extension.

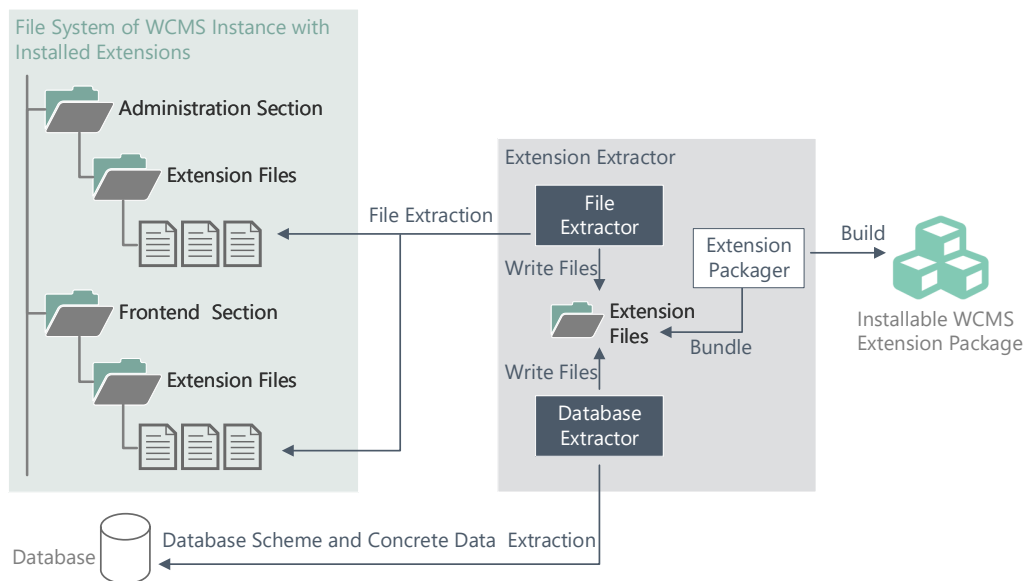


Figure 6.21: Extension Extraction Concept

6.3.2 ExtPorter: A Joomla-Specific Component Extractor

With regard to the evaluation of MDE during our addressed development scenarios, we implemented an actual extension extraction tool for components of the WCMS Joomla (version 3). This tool, called **ExtPorter**, implements the concept which is outlined above. It addresses the extraction of Joomla component files and database schemes as well as actual data from corresponding database tables. The current implementation does not support other extension kinds, since they offer a similar file structure in installable packages and when deployed to a Joomla instance. As mentioned above, the files of a Joomla component are copied to different folders of the Joomla filesystem during its installation. Moreover, components typically represent the only extension kind which manages own corresponding database tables.

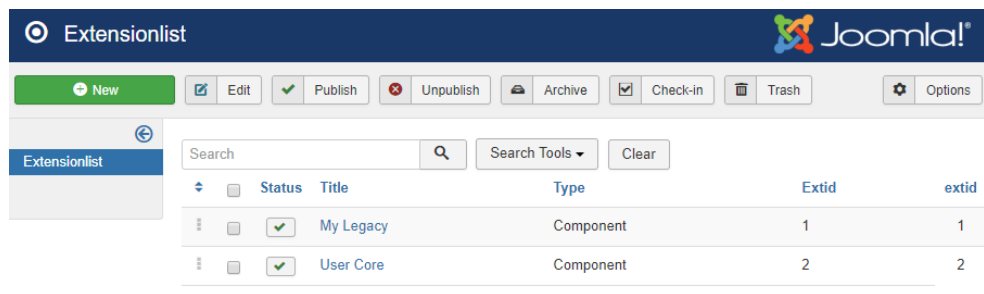
The ExtPorter tool itself is realised as Joomla 3 component. It must be installed to the system where extraction activities have to be performed. This decision has two advantages. First, information of installed extensions can be accessed by using functions of the Joomla framework. So, the implementation effort can be reduced and direct access to the file system or the database is not necessary. Second, Joomla developers can use the tool within their well-known environment. The following extraction process is performed by the ExtPorter component:

1. *File Extraction:* First, the manifest file of the legacy component is parsed in order to gather the corresponding folder structure and language files. Based on the result, the folders and files are copied to a new target folder, following the community guidelines for component file structures (cf. Section 6.2.2).
2. *Database Scheme and Data Extraction:* In the first step of database extraction, queries for database schema definitions (e.g. `CREATE`, `ALTER`, `DROP`) are analysed. To this end, the SQL installation script and all update scripts are investigated successively. All table definitions are temporarily stored in a list, which is constantly updated during the analysis process, in order to consider create, edit or delete operations. Then, all data manipulation queries (e.g. `INSERT INTO`, `UPDATE`, `DELETE`) are parsed and added to the list. The resulting database schema of the component and data manipulation queries are then written into a new installation script file which is placed in the extension folder which was created during the file extraction step. For parsing the SQL files, we use an existing PHP-based *SQL_Parser library* [152] as lexer during syntactic analysis in addition to the *PHP-SQL-Parser library* [226] for the actual parsing process gaining a hierarchical object representation of the SQL queries. Both libraries provided the most robust solutions in comparison to other SQL parsers in the PHP context. However, the *SQL_Parser library* detects more SQL commands, whereas *PHP-SQL-Parser library* is better suited for parsing attributes and types. Therefore, we combine both libraries for the best parsing result.
3. *Extension Packaging:* In the final step, the newly created extension folder is packed to a Zip file, which can be downloaded via the administration section.

Once installed, the ExtPorter component can be used as common Joomla component in the administration section of the hosting Joomla instance. It provides two views for the creation of extracted components. The *details view* can be used for the creation of a new installable extension package based on an installed component (see Figure 6.22). After the packaging process, the component can be downloaded as Zip file which is linked in the view (see Figure 6.23). All extracted components can be found in a *list view* (see Figure 6.24).

Figure 6.22: Details View: Create a new Extracted Component

Figure 6.23: Details View: Installable Component Package as Zip File



Status	Title	Type	Extid	extid
<input checked="" type="checkbox"/>	My Legacy	Component	1	1
<input checked="" type="checkbox"/>	User Core	Component	2	2

Figure 6.24: List View of the ExtPorter Component

6.4 Model Extraction of Legacy Extensions

In order to support development scenarios, which require reverse engineering of existing extension code, *we propose an approach for the automatic model extraction of WCMS extensions*. After introducing a model extraction concept, we present a prototypical implementation for the creation of eJSL DSL instance models by extracting model information from existing Joomla extensions.

6.4.1 Concept

As described in Chapter 4, an MDE adoption during development scenarios in the WCMS domain requires extension models which include features of existing extensions. To achieve this, different model extraction strategies can be carried out. The most tedious strategy is the manual creation of an extension model (cf. problem statement 3). However, by following this procedure, modellers are not biased by implementation details and can focus on the required extension features. In contrast, a fully automated general discovery process (e.g. with MoDisco) typically leads to complex models, which are hard to refine and therefore require an additional model transformation step. We propose a discovery strategy that is based on the exclusive extraction of extension information which can be described by a specific DSL. This may lead to information loss of implementation details, but ensures adequate DSL instance models. These models can be used as basis for manual reengineering actions addressing our stressed development scenarios for augmentation, migration and modernization of legacy extensions (cf. Section 4.2).

Our concept is based on three consecutive steps, taking installable extension packages as input and offering an instance model of the eJSL language as output (see Figure 6.25):

1. *Extension Inspection:* To support various extension kinds, independently of the underlying technology, we propose the inspection of the extension package as first step. Extension packages from legacy extensions can be either downloaded from the WCMS-specific extension directory (cf. Section 2.2.1) or extracted from a running WCMS instance, e.g. based on the concept we proposed above (cf. Section 6.3). The input package should follow the file structure which is required for a clean installation on the target WCMS. This typically includes a deployment descriptor such as the manifest file in Joomla or Drupal. Parsing this file should be sufficient for the determination of suitable file parsers which are invoked during the initiation of the next step.
2. *Extension Parsing:* In the second step, the extension is parsed and translated to an internal representation. To this end, suitable parsers for the required languages must be included.
3. *Model Generation and Evaluation:* In the last step, the internal representation is translated to an instance model of a specific DSL, in our case the eJSL DSL. Furthermore, the internal model can also be evaluated, e.g. in order to determine the applicability for reverse engineering of the extension.

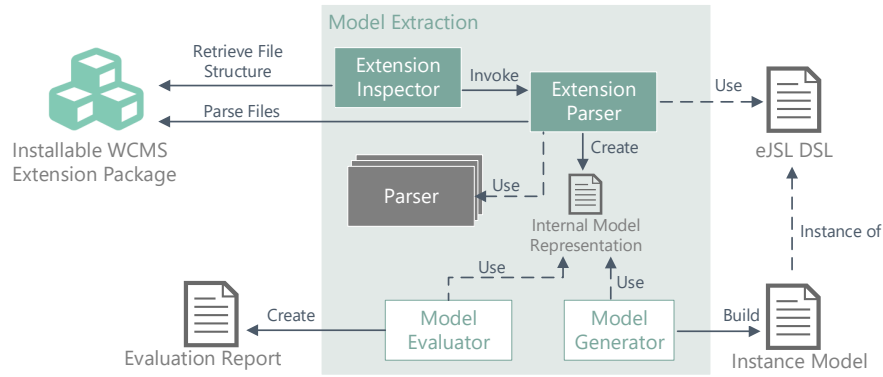


Figure 6.25: Model Extraction Concept

6.4.2 JExt2eJSL: Model Extraction of Joomla 3 Components

In order to achieve various development scenarios, such as augmenting or migrating an existing Joomla extension, we developed a model extraction application named **JExt2eJSL**. So, we provide tool support for decreasing the required effort during reverse engineering of existing Joomla 3 extensions (cf. problem statement 3 and 5). The tool has been developed as Scala application, which consists of domain-specific parsers for the various technologies which are used during Joomla extension development. As part of a master thesis in 2019, the tool was refactored and extended in the context of researching the discoverability of MDE compatibility of legacy applications, prototypically for Joomla 3 extensions. Therefore, it parses existing Joomla 3 extension packages and creates representing eJSL models. Whereas, the tool is able to create extension models for all supported Joomla extensions, the most effort went into support for components, which are the most complex extension kind. Joomla 4 extensions are currently not supported. Scala was the language of choice, since a set of sophisticated parser combinators exist, e.g. the standard scala parser combinator [207] and FastParse [85]. So, the implementation effort of required parsers could be reduced. Moreover, the decision provides a straightforward integration of the application into a Java-based development environment, since it runs on the Java VM. We provide different possibilities to use the tool: It can be used as independent GUI, as Eclipse plugin, or by a CLI command. The latter allows a direct integration into an automated test environment, e.g. as part of a continuous integration, delivery, or deployment (CI/CD) pipeline. Moreover, we incorporated the model extractor homogeneously to the web IDE which also includes the eJSL editor and the code generator for Joomla extensions (cf. Section 5.4). Figure 6.26 illustrates the implemented architecture of the JExt2eJSL application. This architecture adheres to the model extraction concept which is mentioned above.

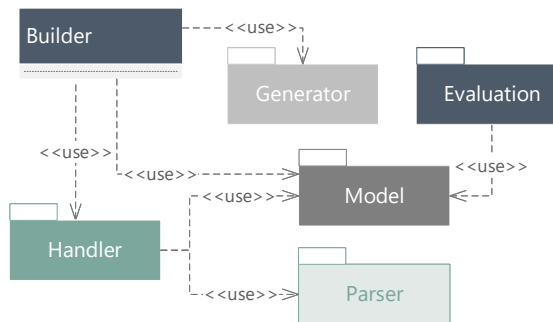


Figure 6.26: Overview of the JExt2eJSL Architecture

A general overview of the implemented transformations during the extraction process with JExt2eJSL is illustrated in Figure 6.27.

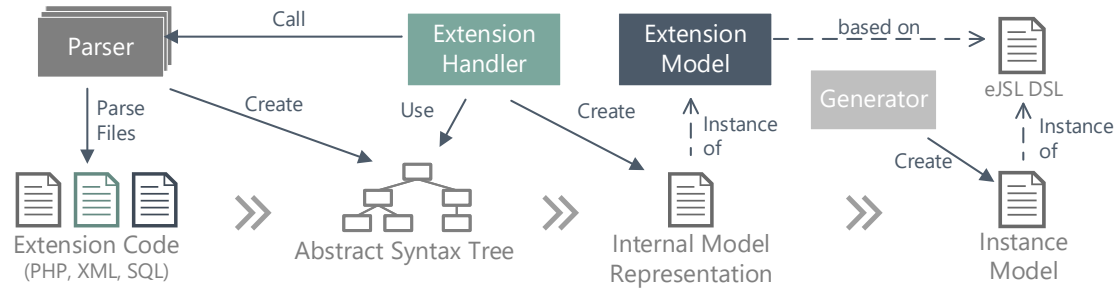


Figure 6.27: Extension Parsing Process

Due to different development approaches for Joomla extensions, JExt2eJSL supports various file structures of an extension package as input during *extension inspection*. To achieve this, the **builder** class analyses the manifest file of the input extension which includes the folder structure of the extension package. Based on the result, the extension kind and file structure is dispatched to a suitable handler class, which creates an internal representation of the extension.

The *extension parsing* is performed by the invoked **handlers**, which extract actual extension information. To extract as much model information as possible, the implemented handlers require suitable parsers during the extension extraction process. So, we can operate on the Abstract Syntax Tree (AST) [19] in order to extract relevant information from the source code. Based on the gathered extension information, the handlers create the internal extension representation in the style of the eJSL DSL. This can be achieved by using **model** classes which include the DSL features which are definitions of **entities**, **pages** and **extensions**.

We included existing up-to-date parsers for SQL and XML, and implemented a new parser for PHP which can be integrated into a Java-based project. For parsing SQL statements, we use the JSQParser [255] library which supports MySQL and transforms all kinds of SQL queries to a traversable Java class hierarchy. The current version of JExt2eJSL exclusively parses data definition queries, since the eJSL DSL facilitates no support for actual data. Additionally, we include the stand-alone but standard Scala XML library [86] for processing XML files such as the manifest and field descriptor files. Current existing PHP parsers are not up-to-date. Especially PHP parsers which can be used in Java are clearly outdated. Therefore, we implemented a general PHP parser which can be used for every kind of PHP project. It is based on the current PHP 7 language specification and can be used as independent parser within Java and Scala projects. The parser was initially developed as part of a bachelor thesis and published as open source project on GitHub: <https://github.com/thm-mni-ii/PHP-Parser>.

To extract a model as complete as possible, the tool comprises various extraction strategies. First, the extension metadata is extracted from the manifest file and all provided language files are parsed. The result is translated to an **extension** definition as it is designed in the eJSL DSL. Whereas the metadata can be extracted from the parsed manifest itself, the actual translations are extracted from the referenced language files and are added to the language definition as key-value pairs. If the input extension is a Joomla component, gathering the required page references requires a more complex translation and depends on the discovery results of **page** information (processed by the **ComponentHandler**). In the first step of the page discovery, view classes are parsed and references to a model instance are searched. Then, the corresponding model class is parsed in order to gather information about the page kind. If the model class extends the **ListModel**, **JModelList**, or **JModelLegacy** class, the corresponding page is translated

to an index page object. Though, if the model class extends the `ItemModel`, `JModelAdmin`, `FormModel`, `JModelItem`, or `JModelLegacy` class, the page is translated to a details page object. Other model classes are not supported. Based on the page kind, special handler classes are instantiated (`IndexPageHandler`, `DetailsPageHandler`), which extract the respective required page information. To achieve the full extraction of e.g. form fields which are translated to table columns of a list page, various files such as the view class file as well as the field and the form description files are parsed and investigated. Moreover, all configuration descriptor files are parsed and analysed, to extract parameter definitions which are translated to custom **parameters** in the model representation. If no model information can be extracted, a custom page object is created. This page kind is provided by the eJSL DSL to allow individual page definitions (cf. Section 5.2.2). The gathered page information is then added to the extension part of the component model, whereby frontend and backend pages are reused if they have the same name and have references to the same entity.

Then, the handlers extract **entity** information by parsing SQL files (installation and update scripts), table classes, the DAO (e.g. a model class in a component or a helper file within a module), and data definition and manipulation statements within a view implementation. So, all possible entity definitions are considered during the extraction process. Whereas the tables are transformed to entity objects, table attributes are translated to entity attribute objects including database attribute properties such as default values. In the latter case, database types are mapped to the supported eJSL standard type kinds (cf. Section 5.2.1) based on the mapping as presented in Table 6.3 on Page 134. If a type is found which is not supported by the eJSL DSL, new data types objects are created. This is supported by the DSL in order to allow individual types definitions. Foreign key definitions SQL files are transformed to entity references.

After extracting model information by the handlers, *model generation and evaluation* can be performed. The **generator** part includes code templates for the creation of eJSL instance model files. These templates adhere to the concrete syntax as defined in the eJSL grammar. Missing model elements are translated to placeholders (cf. Listing 6.11).

```

1  links{
2      InternalContextLink index {
3          target = <Refer to a target here>
4      }
5  }
```

Listing 6.11: Placeholder within an Extracted Extension Model

So, modellers can identify incomplete model parts which require a manual refinement. If an extracted **page** cannot be assigned to a page kind, a **custom page** is created in the eJSL model. This allows that the model can be used as input for an extension code generation, even though the page must be further reviewed and refined by the modeller.

A drawback of the tool is based on its tailored implementation for the Joomla WCMS. It necessitates input files which adhere to the architectural guidelines of the Joomla core. Inadequately implemented extensions lead to inconsistent models or errors during the model extraction process. Individual file structures are supported, if they are adequately specified in the manifest file. However, if none of the mentioned classes are extended, e.g. in a respective model class, pages cannot be resolved. The extraction tool is strongly based on the eJSL DSL. The intention behind that is the straightforward support during the stressed development scenarios in this work based on the eJSL DSL features (cf. Section 5.2). So, individual classes and methods cannot directly be extracted. Though, by using a library as input extension allows the transformation of whole packages, classes, methods, and attributes from code to model level. If the extension mainly consists of individual file structures or does not follow common coding standards, a manual model extraction strategy should be applied or the extension should be reengineered.

Another limitation of the current implementation of the JExt2eJSL tool is the lack of reports, concluding the amount of extracted information. However, as Figure 6.26 illustrates, we included evaluation features by providing a metric which determines the amount of code which can be reverse engineered in order to indicate the ability for reverse engineering of the input extension. So, developers can decide, if an automated extraction by using the model extractor is a reasonable strategy. However, the evaluation part of JExt2eJSL is not relevant in the context of this work and will therefore not be further considered.

6.5 Evaluation

In this concluding evaluation section, we discuss the results of this chapter with respect to the specified MDE infrastructure requirements for code generation and reverse engineering of legacy extensions which are defined in Section 4.3. Moreover, we demonstrate the Joomla-specific generator concept implementation with three case study examples. Finally, we discuss the technical adequacy of the implemented tools addressing quality assurance and presenting the results of scalability tests of the code generator during runtime.

6.5.1 Requirement Verification and Validation

Analogous to the requirements verification and validation of the DSL and corresponding editors (cf. Section 5.6), we examine the coverage of the transformation tool requirements (Section 4.3) by the presented tools in this chapter. To this end, we examine the coverage of the acceptance criteria for the generator (Requirements R3.1-R3.5) and reverse engineering facilities (Requirements R4.1 - R4.2). Additionally, we discuss the general infrastructure requirements coverage for the presented MDE infrastructure facilities (R5.1-R5.3).

Subject: Code Generator

The main Requirement R3.1 for the code generator is fulfilled, if the *generation of full installable extension packages* is provided. Due to the data-intensive operations of WCMS extensions, these packages must comprise complete CRUD functionality for managed data entities. Based on the extension kind to generate, this includes the generation of adequate file and code structure such as the implementation of MVC for Joomla components. Moreover, the features presented in Section 2.2.2 must be covered by the generator. These features include, among others, internationalization/localization by language files, interoperability between extensions, backend/frontend support, and core support. These requirements are fulfilled by the presented code generator concept and implementation for Joomla 3 and 4. As presented, the generator creates installable Joomla extensions based on the community guidelines for extension development. The generated extensions adhere to the coding styles and architectural standards which are provided by the Joomla community. Thus, the generator implementation covers the acceptance criteria for Requirement R3.2.

Generated extensions include common extension features for the specific extension kind. To this end, a set of adequate reference extensions were investigated during the development of generator templates. To evaluate the generator, we used various eJSL instance models comprising the different aspects of the language in order to reach high coverage of generator features. Additionally, we used the generator in combination with the eJSL model editors during **three case study examples** within the Institute for Information Sciences². The institute developed and maintained extensions for the WCMS Joomla throughout the last decade. These extensions are deployed to several Joomla installations which represent the websites for the university of applied sciences in

²Institute at the computer science department at the Technische Hochschule Mittelhessen, <https://www.thm.de/mni/forschung/institute-gruppen/ii/>

Gießen, Germany, (Technische Hochschule Mittelhessen, THM) and its departments. Whereas in the past, most of the extensions have been developed in the conventional way, we used the MDE infrastructure for the development of new extensions or the augmentation of existing ones. In the following, we present three case studies of how we used the tools for the initial development of components in administrative and development activities, showing that the implemented MDE infrastructure is ready for being used in realistic projects. These case studies have also been presented in [190].

Pre-Course Management for Students: At THM, students can attend a pre-course before their regular studies. This allows them to prepare for their study programme. The management of pre-course students has been done by an outdated external website, which was not part of the university's official website pool. Therefore, the requirement was to incorporate the pre-course management into the main Joomla installation of the university. To this end, we created an eJSL instance model including the pre-course requirements and used the presented code generator for the initial development of a Joomla component. This component can be used to manage pre-courses and their attendees within a Joomla instance, since it provides CRUD views for the management of courses (cf. Figure 6.28 and Figure 6.29) and registered attendees. Additionally, the component works together with the user management of the Joomla core.

	<input type="checkbox"/>	Status	Coursename	Start	Finish	Attendance	Capacity	ID
...	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Mathematik	2016-02-10	2016-02-18	0	100	1
...	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Deutsch	2016-02-09	2016-02-23	0	100	8
...	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Programmierung	2016-02-09	2016-02-23	0	100	9

Figure 6.28: List View for Pre-Course Management

Course

Course

Permissions

Coursename

Mathematik

Start

2016-02-10

Finish

2016-02-18

Semester

Sommer

Capacity

100

Attendance

0

Attendancewaiting

0

Figure 6.29: Edit/Details View for Pre-Course Management

The defined model consisted of 9 data entities and 18 pages, whereas the generated component comprised 30 pages in form of MVC combinations. Since these combinations require the largest amount of code, the high number of pages in the model led to ~25k LoC for the whole component with only 347 LoC in the corresponding model. The generated extension was refined by individual functionality (~2.5k LoC) before it was installed and used productively. Thus, more than 90% of the component could be created in a model-driven manner. The missing 10% were custom

business logic which had to be incorporated by hand. Though, this was easily accomplished by adding new functions to the controller and model files as well as changing HTML definitions in a view file.

The initial development process was part of a student's final project in 2016. The component was then used productively at the THM. However, the features of the component became incorporated to another Joomla component (THM Organizer [9]) which is used for the whole resource management like rooms, courses, and schedules of the university within a separate Joomla instance [231]. Therefore, we refined the model to provide it as showcase model as already explained in Section 5.5.

Joomla Extension Exporter: Another extension created with the generator is the previously described ExtPorter component (cf. Section 6.5.1). Once installed, the component creates an installable extension package by inspecting installed components of a Joomla instance and copying all files into an installable package in the correct file structure. So, developers are able to create installable packages of legacy components, e.g. to use them as input for the presented model extraction tool JExt2eJSL (cf. Section 6.4.2). Even though the component has some individual parts, the main structure and the management views of the component have been fully generated by the presented code generator. The defined model consisted of only one data entity and two pages which are used twice (frontend and backend) and had a total amount of 83 LoC. The generated component with 4 views resulted in ~5k LoC.

Besides the use of the generator in actual projects, we use it for *teaching purposes* in web development lectures. In these lectures, students have to learn how to develop PHP-based applications. To this end, they have to (further) develop extensions for Joomla or create new features/patches for the Joomla core. In the first years, the students required most of the time for learning the structure of Joomla extensions and how to implement them. By applying the model-driven approach, we were able to reduce the learning effort. By using the eJSL model editors and the Joomla-specific code generator, the students get a better understanding for the required file and code structure, since they can easily change some abstract parts in the model, generate extensions anew, and inspect the changes.

Moreover, the generator was used within the quantitative and qualitative analysis of MDE of WCMS extensions within conducted experiments. During the experiments, participants had to create full installable extension packages for the Joomla WCMS in order to apply the addressed development scenarios which are described above (cf. Section 4.2). The experiments are presented in more detail within Chapter 7 below.

Our generator provides *partial code generation* which covers the acceptance criteria for Requirement R3.3. For each main part of the language (entities, pages, extensions), partial code fragments are generated. Relations between legacy and new code fragments can be considered by using the generator hook feature which is provided by the eJSL DSL (`@preserve`). The generated files can be used for updating a deployed extension without using the update mechanism of the Joomla system. New files can directly be copied to the respective extension folder within the file system of the Joomla installation. This requires access to the file system, which is not typically guaranteed. The file structure for the partial extension code adapts the structure of installable extensions but does not contain extension-specific files like the manifest or entry file. Figure 6.30 illustrates the different file structures of an installable component and generated partial code. For each referenced page in the eJSL instance, MVC and database-specific files are generated.

Support for smart and dummy models (cf. Requirement R3.4) is ensured by the proposed generator frontend. This frontend includes a resource transformer which performs a model preprocessing step during each execution of the generation process. During this step the model is annotated with default values for model features which are not specified but required during the actual generation process.

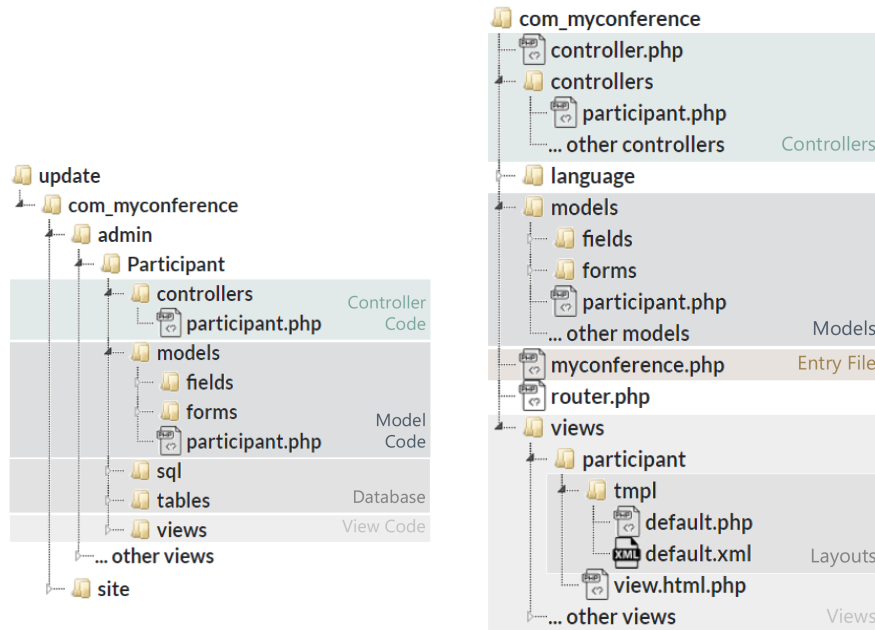


Figure 6.30: Generated File Structure of Partial Update Code (left) and Installable Component Code (right)

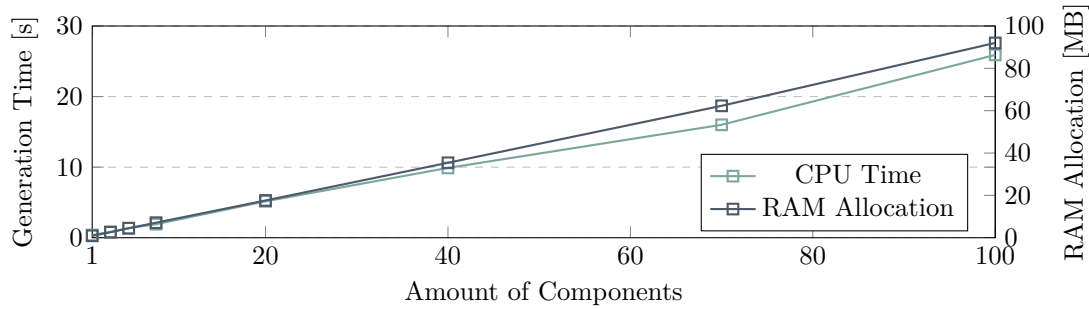
In order to cover *custom code support* (Requirement R3.5), dummy code is generated for each part of an extension which requires a manual refinement. However, if custom code is added to generated code, it may be overwritten during further re-generation actions. Therefore, we propose to make use of sub-classing in order to add custom features. Alternatively, developers have to isolate refined files and assemble generated and custom files by hand. Based on the addressed development scenarios and the extensive CRUD functionality of WCMS extensions, the requirement had low priority during the development of the generator and is therefore not completely fulfilled. During further research and refinements of the generator, custom code support should get more attention. A promising approach could be the adaption of custom code models which can be referenced by eJSL model instances.

To assure *generator quality* (cf. Requirement R5.1) we test the expected *functional correctness* of generated extensions manually for both supported Joomla versions. After generator refinements, we install generated extensions based on our showcase models (see Section 5.5) to a suitable version of a Joomla instance (version 3 or 4) and test them by applying manual End-to End (E2E) tests. We test the (de-/re-)installation routine, language translations (multi language features), extension parameter features and configuration, the whole CRUD behaviour of managed entities, dependencies between extensions (e.g. between modules and components), and correct interaction between views (link behaviour). Unit tests for automated generator tests are not implemented (during earlier stages of the generator development, test cases were implemented but not further updated or maintained). Though, in order to support infrastructure developers by automatic testing, we added test models and a main test class to our generator project. This class can be used in order to implement functional tests which can be processed by the current JUnit 5 testing engine [20]. Testers can use the test models as input and define parts of Joomla extension code as expected output of the code generator as test cases. These tests can then be executed after each generator refinement, e.g. automatically as part of a CI/CD integration strategy.

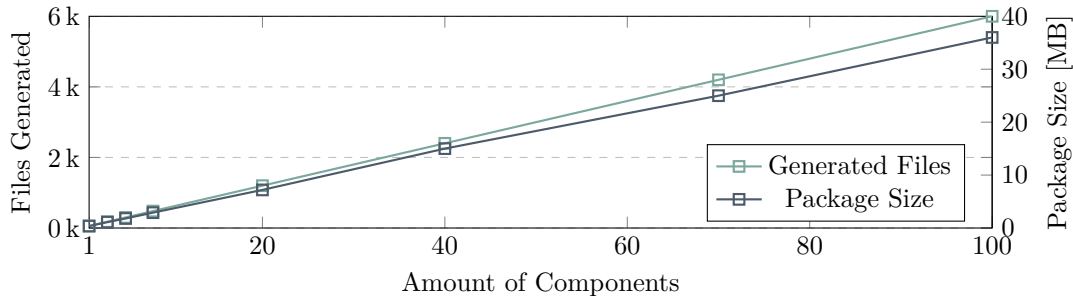
Additionally, we ensure syntactical correctness of generated extensions after each generator refinement by using supportive tools like the PHPCodeSniffer [220], PHPMD (PHP Mess Detector) [180], and JSLint [51]. In order to keep up with the community guidelines, we include the PSR-12 [179] and Joomla [111] coding standards which can be included in the PHPCodeSniffer.

In addition to functional tests, we measured the execution time during code generation in order to test the *scalability of the generator*. We used eJSL instance models of different sizes with different complexities in order to find performance drivers. We performed five scalability tests, whereby we scaled different model features. All models consist of at least one component which includes at least one list and one details page in the backend and the frontend section. For each entity in the models, we added another pair of list and details pages which were added to both sections of the component. In total, we used 38 different instance models which scale up to 5400 LoC for the most complex model. During each test, we executed the generation process five times and profiled the average generation time, allocated RAM, number of generated files, and size of the generated extension package. All tests were executed on a machine with an Intel Core I7 8750H 2.21 GHz and 32GB RAM. For the profiling we used JProfiler [61] (version 11.1). In order to measure the amount of generated files and their cumulated file size, we executed the `"find ./ -type f | wc -l && du -sh ."` command within the root folder of each generation result. All measurement concepts, settings, and results are collected in Appendix D.

Scalability Test 1 (Components): In the first test, we measured the impact of a scaling component count within an eJSL instance model. We created a component with one list and one details page which refer to one entity with four attributes and no references. During the test, we increased the number of components in the model by duplicating the component. The pages and entities were not changed during this test. According to the results of the test (see Figure 6.31), all measured variables grow in a linear way. The generator required 26 seconds and allocated 92 MB RAM for the generation of 100 components with code for 4 views within 2 sections (6000 files with a complete file size of 36MB).



(a) CPU Time and RAM Allocation



(b) Amount of Generated Files and Package Size

Figure 6.31: Scalability Test 1 (Components)

Scalability Test 2 (Pages): The second test measured the impact of scaling list and details pages within one component definition. During each test iteration, we added a new list and details page to the model and created a reference to both pages in the backend and frontend section of the component definition. To avoid any measurement biases, we decided to use the same entity as entity reference within all pages. This entity consist of four attributes but has no references. Figure 6.32 summarizes the test results. We measured a generation time for a component with 100 page pairs which are generated in the backend and frontend section of ~ 56 seconds with a memory allocation of 87 MB. In total, 2535 files with a package file of 19 MB were generated.

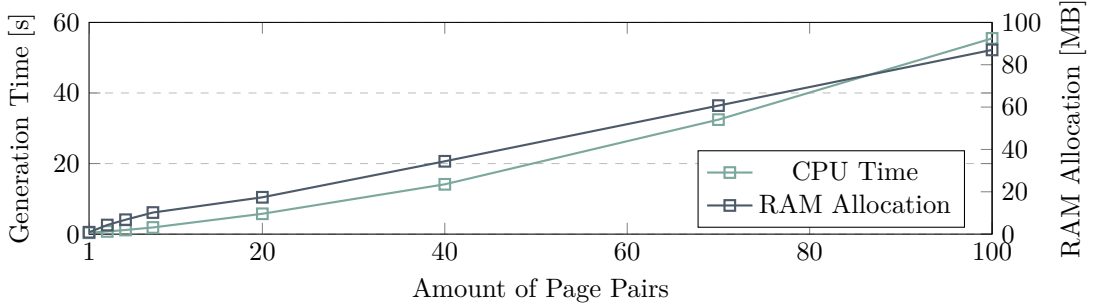


Figure 6.32: Scalability Test 2 (Pages)

Scalability Test 3 (Entities with References): Another test profiled the effect of a scaling number of entities with references to other entities. We specified 1 entity without references and successively added entities with a reference to this same entity. For each new entity, a new list and details page was added to the model. For each new page pair, we added a reference to both in the backend and frontend section of one entity. The largest model comprised 404 page references in the component. The test results show that the generation time and allocated memory grew similarly (cf. Figure 6.33), whereas we measured a generation time of 162 seconds and 198 MB RAM with the largest input model (3062 files, 21 MB package size).

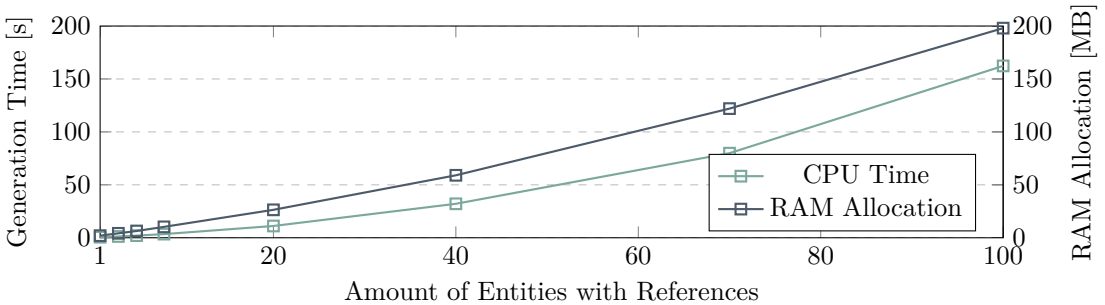


Figure 6.33: Scalability Test 3 (Entities with References)

Scalability Test 4 (References in one Entity): In contrast to the previously described test, we measured the effects of an increasing number of references within one entity definition. We increased the number of entities without references and created a new reference in the same entity to all of these new entities. Moreover, we added a list and details page for each new entity and created a reference to all page pairs to the backend and frontend section of one component. The most complex input model consists of 100 entities without references, 1 entity with 100 references, 101 page pairs, and 1 component with 404 page references. During the test we observed a quadratic growth of the generation duration, whereas the RAM allocation grew linearly (cf. Figure 6.34). The most complex model led to a generation time of 204 seconds and a memory allocation 139 MB. In total 3062 files were generated (22 MB package size).

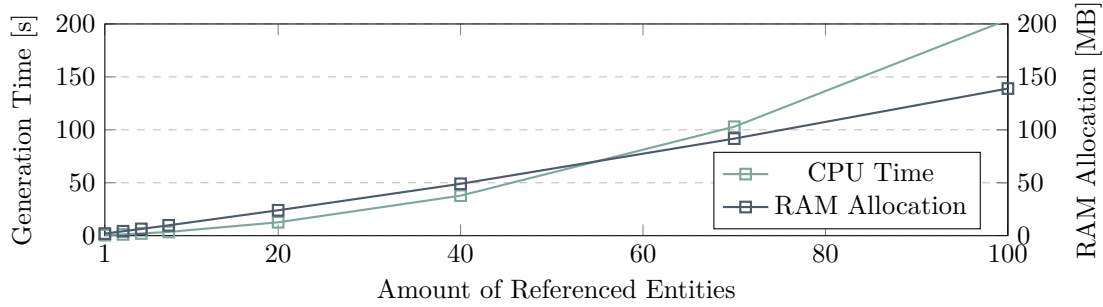
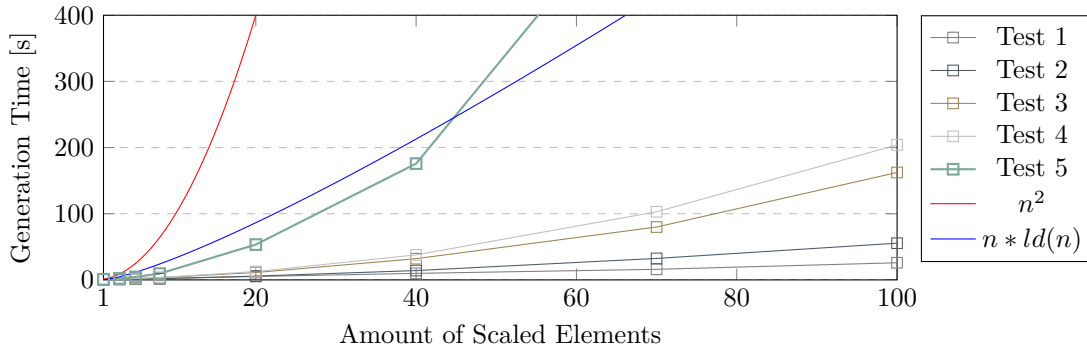
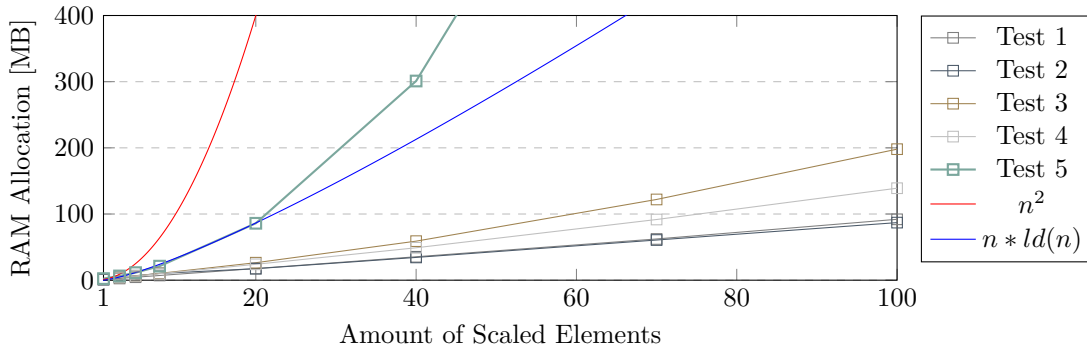


Figure 6.34: Scalability Test 4 (References in one Entity)

Scalability Test 5 (Entities with Many-to-Many References): With the last test, we measured the effect of scaling entities with many-to-many references. This is the most complex model feature with the most complex generator logic due to the implicit translations to suitable database scripts and views (cf. Section 6.2.2 above). Similar to the previously described test, we increased the number of entities with references to one specific entity and added a new reference to this entity back to the new entity within each test iteration. During this test, the generation time and allocated memory grew quadratically (cf. Figure 6.35). In comparison to the other tests, we observed a more critical data for all measured variables. The most complex model consists of 101 entities with $1..*$ references, whereby entity1 had a reference to each other entity, which in turn had 1 reference to entity1 each (cf. Figure D.5 in Appendix D). This model led to a total generation time of 18.5 minutes with 1.8 GB memory allocation (3762 files, 29 MB package size).



(a) Generation Duration



(b) RAM Allocation

Figure 6.35: Scalability Results of all Tests

In summary, the tests show that the code generation process for Joomla components is not critical considering the scalability aspect. According to the results as presented in Figure 6.35 we extrapolate a quadratic runtime and memory allocation $\mathcal{O}(n^2)$.

In addition to the quality assurance requirement, the *integration of the generator into existing development processes* (see Requirement R5.2) is implicitly ensured by using Xtend for the actual generator implementation. In combination with the Xtext framework which is used for DSL definition, plugins for the Eclipse IDE and IntelliJ IDEA can be generated straightforwardly. These plugins comprise the whole MDE infrastructure facilities for applying forward engineering of WCMS extensions. Furthermore, a similar plugin for the PhpStorm IDE is provided and the Joomla-specific generator is integrated into a web IDE, together with the model editor for the eJSL DSL (cf. Section 5.4) and the model extraction tool JExt2eJSL (cf. Section 6.4.2). All plugins can be generated automatically by executing automation scripts (Gradle) which are provided as part of the JooMDD project (see Figure 6.36). These scripts include various tasks for the automatic build of the JooMDD plugins and plugin repositories (or update sites) for all supported IDEs (cf. the JooMDDDeploy task in Figure 6.36). This also includes tasks for increasing a respective version number of the plugins in all necessary description files of the project. The same applies to the web IDE which is assembled and can be deployed to a web server automatically (cf. the jettyRun task in Figure 6.36). So, the whole infrastructure development can be part of a CI/CD or even DevOps strategy which publishes new plugin versions within short release cycles automatically. This aspect is a relevant driver in software development nowadays (cf. [93], [241] and [270]) and is therefore considered in order to increase the maintainability of the MDE infrastructure. To make use of GitHub Actions [78] for task automation in GitHub, we currently work on the incorporation of the Gradle tasks into a CI/CD pipeline which will be triggered automatically with each merge of a pull request into the JooMDD project. So, we will enable distributed development within the open source projects.



























>  application	
>  build setup	
>  build	
>  distribution	
>  documentation	
>  help	
>  ide	
>  intellij idea	
>  intellij	
>  jetbrains_wizard_help_task	
>  joomdd_help_task	
▼  joomdd_live	
 IntelliJWithEJSL	Create eJSL, add Xtend and run IntelliJ IDEA
 JetBrainsIDEWithAllPlugins	Create Wizard and eJSL, add Xtend and run JetBrainsIDE
 JetBrainsIDEWithWizard	Create Wizard and run JetBrains IDE
▼  joomdd	
 JooMDDDeploy	Create all deployment resources (JetBrains IDE) , prepare Eclipse Deploy
 incrementMajorVersion	Increment the Major Version for the Project
 incrementMinorVersion	Increment the Major Version for the Project
 incrementPatchLevelVersion	Increment the Major Version for the Project
 regenerateXtendGenFolder	delete and create xtend gen
▼  run	
 jettyRun	Starts an example Jetty server with your language
▼  upload	
 uploadArchives	Uploads all artifacts belonging to configuration
>  verification	

Figure 6.36: Gradle Tasks for Build Automation of JooMDD Plugins

Additionally, tasks for testing purposes are also included. These tasks create the required IDE plugin and starts an IDE instance which already includes the plugin. So MDE infrastructure developers can test the current version without any installation overhead (cf. *joomdd_live* in Figure 6.36). With the provided tasks for automated build and deploy actions, the acceptance criteria of Requirement R5.3 is fulfilled.

Subject: Reverse Engineering Facilities

The reverse engineering facilities which are presented in this chapter comprise a *model extraction* tool for installable Joomla extension packages as well as an *extension extractor* for deployed Joomla components which creates such installable extension packages. So, we address both main requirements R4.1 and R4.2. The model extraction tool, JExt2eJSL discovers extension information and creates an eJSL model instance based on the provided feature of the extension package. If a deployed extension has to be used for model extraction, the ExtPorter component can be used in order to create the necessary folder structure for installable packages. In order to support the stressed development scenarios exemplarily for the Joomla WCMS, the model extraction tool is currently limited to the Joomla WCMS. Moreover, the tool is strongly tailored to the eJSL DSL. It requires extension packages which strongly adhere to the architectural guidelines of Joomla. Inadequately implemented extensions are not supported, leading to inconsistent models or exceptions during the use of the tool. Individual classes and methods cannot directly be extracted, since the tool only considers special MVC classes which implement the extension API of the Joomla core. However, by supporting the extraction of library extensions allows to transform packages, classes, methods, and attributes to eJSL model definitions. Legacy extensions which mainly consist of individual file and code architectures but have to be considered during the stressed development scenarios (scenario 2-5), cannot be inspected by our proposed tool. For this case, we propose to reengineer the extension before using the tool or apply a manual reverse engineering process. In future work, a more general reverse engineering approach should be researched, based on the presented concepts as presented in Section 6.4.1.

In order to ensure the *quality of the RE facilities* (cf. Requirement R5.1), we test the correct behaviour of both implemented tools manually. We use generated extensions created with the presented code generator based on the provided showcase models (cf. Section 5.5) as input for the model extractor and compare the resulting model with the original showcase model. In first development iterations, we used a set of popular third-party extensions from the extension directory of Joomla as input packages. However, we observed, that *most of these extensions require a reengineering since they do not adhere to the architectural guidelines of the community* (cf. problem statement 1). Based on this insight, we implemented the extension inspection as first step in order to allow a more flexible architecture. The ExtPorter component tool was tested by extracting installed third-party components which were compared to the original extension packages. So, we were able to identify and integrate special file structures which have to be considered during the extension process. In contrast to the generator implementation, no scalability tests were executed.

Addressing the *integration of the RE tools into the development process* of extension developers (cf. Requirement R5.2), we followed different strategies. The extension exporter has to be used as component within a Joomla installation on which the target extension is deployed. This allows all administrators to extract an installable extension package in their well-known environment without the necessity of using an additional application. However, the ExtPorter component must be installed to the Joomla installation. This can typically only be performed by administrators. This can be a drawback of the decision. In contrast, the model extraction tool is implemented as stand-alone application, which is also provided as Eclipse plugin, and can also be used by a CLI command. This allows the integration, e.g. into the web IDE where it is provided together with all the other presented MDE facilities. Moreover the tool can be included to a CI/CD pipeline which build the plugins and executable application file (cf. Requirement R5.3).

6.5.2 Threats to Validity

In this chapter, we presented transformation tools for applying forward and reverse engineering of WCMS extensions with prototypical implementations for the Joomla WCMS. These tools extend the MDE infrastructure for WCMS extensions addressing the requirements which are presented in Section 4.3. This includes a code generator for Joomla 3 and 4 extensions, as well as an extraction component for deployed Joomla components, and a model extraction application which creates an eJSL instance model based on an existing Joomla 3 extension package. Similar to the *internal validity threat* as presented in the previous chapter, we did not involve external extension developers during the evaluation of the presented transformation tools. So, early adoptions of design decisions or even the presented concepts may have affected the latest versions of the implemented tools which are incorporated in the JooMDD infrastructure. Furthermore, the quality is not evaluated properly during each iteration, since no adequate unit tests for automatic testing exist. The same applies to scalability tests for the RE facilities. However, in the next Chapter, we present the results of a quantitative and qualitative analysis of applying MDE in the WCMS domain with external developers. During a set of experiments, a late version of the JooMDD infrastructure, including the presented transformation tools, was used during the application of common development scenarios. Defects which were found during these experiments were fixed and new feature request were implemented afterwards.

In order to ensure *external validity* of our approach, generators and RE facilities for additional WCMSs should be implemented. So, the universal applicability of the eJSL DSL and the presented concepts for the transformation tools can be demonstrated. This, however, goes beyond the bounds of this work and should therefore be addressed as part of further work. To this end, we propose the implementation for popular WCMSs such as WordPress and Drupal. Moreover, we suggest to evaluate the concepts also for domain-specific WCMSs, e.g. used for the implementation of shop systems. Worth mentioning is the project of Cabot which proposes a DSL and corresponding code generator for the WordPress WCMS [37, 36]. The proposed DSL shows similarities to some parts of our proposed eJSL DSL. However, in contrast to our proposed MDE infrastructure, the DSL and generator for WordPress provides less language features and is not able to handle extension dependencies. Moreover, sophisticated extension kinds such as the ones supported by the Joomla WCMS are not considered. According to first investigations, all language features are covered by the eJSL DSL. The same applies to the code generator concepts. So, an integration of the proposed project into our work seems to be a promising step as part of future work.

7

MDE of WCMS Extensions - Quantitative and Qualitative Analysis

Since software engineering is in its adolescence, it is certainly a candidate for the experimental method of analysis. Experimentation is performed in order to help us better evaluate, predict, understand, control, and improve the software development process and product.

– Victor R. Basili, Richard W. Selby, David H. Hutchens in [18]

For the purpose of investigating the profitability of MDE during WCMS extension development, we presented an MDE concept for the confirmed development scenarios in the domain (cf. Chapter 4). Based on this concept and corresponding requirements, a DSL for WCMS extensions as well as concepts for suitable transformation tool are presented in the previous chapters (Chapter 5 and Chapter 6). The latter also includes prototypical implementations addressing extensions for the Joomla WCMS. In this chapter, we use quantitative and qualitative methods to research the effect of an MDE infrastructure during WCMS extension development. So we address *RQ2: To which extend can MDE support WCMS extension developers during development and maintenance of WCMS extensions?*

First, we present the design and results of a *controlled experiment with WCMS extension developers*. This experiment was conducted in order to quantitatively compare conventional with model-driven extension development during development scenario 1 and 2 (cf. Section 4.2). We discuss the results with regard to the impact of our presented MDE infrastructure on the development speed and software quality during extension development. We also provide insights into the extension development by inexperienced developers using our presented MDE infrastructure. Additionally, the design and observations of a *hands-on tutorial* with industrial practitioners from the Joomla community is presented in this chapter. During a workshop, the participants applied our presented MDE infrastructure during development scenario 1-3, in a semi-controlled manner. So, we achieved qualitative results to allow conclusions on the general suitability of an MDE approach in the WCMS domain.

Excerpts from this chapter have already been published in [185] and [190]. The used documents during the empirical assessments (Appendix E, Appendix G) are also published as online appendix in [189].

7.1 State of the Art

In the past, several efforts have been invested to investigate the impact of MDE in practice. Studies which focus the *practical and industrial adoption of MDE* in the embedded systems or mobile development domain are presented in [258], [32], [139], [244], [216], [142], and [245]. Sousa et al. [216] present the adoption of MDE in an industrial modernization scenario, whereas Whittle et al. [258] introduce a taxonomy of tool-related considerations based on empirical data stemming from industry. The latter distinguishes technical factors (concerning technical aspects of MDE tools) from organizational and social ones (focusing on tool use and application within working processes). This taxonomy was used to analyse interviews from industry, mainly at companies such as Ericsson and Volvo. Although developed for empirical studies in other

domains, most of their lessons learned are confirmed by our studies which are presented below (cf. Sect. 7.4).

Mohagheghi et al. [156] reflect the adoption of MDE in four cases from companies in different domains (enterprise applications, telecommunication, aerospace crisis management systems and geological systems) based on interview and questionnaire studies, focusing on the practical motivation for using MDE and subjective usability aspects. Karg et al. [126] analyse the adoption of MDE in the openETCS project (railway domain) based on practical experience, surveys, and interviews. The authors of both studies state that MDE can be generally applied successfully, affecting developers positively. However, they also mention that methodologies and tools are a main inhibiting factor. An additional work in this context is presented by Baker et al. [16]. The authors report of successfully applying MDE for 15 years at Motorola. As a result of the MDE adoption, the authors report of 2.3 times less required effort, 1.2-4 times less defects, and 2-8 times greater productivity.

In addition to practical and industrial application, several *experimental investigations* have been conducted in various domains in order to compare MDE with conventional development methods. Bunse et al. [31] present the results of 15 three member teams adopting MDE during the development of components in the embedded systems domain. They came to the conclusion, that MDE reduces effort and improves the reuse and quality of software artefacts in the domain. Similar studies were conducted in the web domain. The study which was conducted by Fernandez et al. [65] investigated the usability of web applications being developed in a model-driven way, whereas the closest studies are presented in [147], [176], and [177]. Martinez et al. [147] investigate the maintainability of web applications. The authors compare model-driven development of web applications with code-centric development. They conducted an experiment with 27 graduate students, who had to perform a series of maintainability tasks in two groups. Specifically, they investigated the effectiveness, efficiency, usefulness, and ease of use of the development approaches w.r.t. changeability. As result of that study, the authors found *a perceived loss of control with MDE approaches, that model-driven development is slightly more learnable and less complex than code-centric development, and that developers are not as satisfied with the MDE approach as expected*. Panach et al. [176] compare model-driven development with conventional development of web applications. The authors research the impact of MDE on quality, effort, productivity, and developer satisfaction by conducting a controlled experiment with 26 students as part of an MDD course. The same applies to the presented study of Papotti et al. [177]. The authors compare the performance of conventional development with MDE of web applications by conducting a controlled experiment with 29 senior students. Both studies observed a positive effect on the researched variables during MDE adoption (e.g. 90% reduced development time [177]).

To the best of our knowledge, *there is no other empirical study researching the impact of an MDE adoption in the WCMS domain (Problem Statement 6)*. Therefore, we aim to conduct an experimental investigation to extend the results of the most cited benefits of MDE with respect to quality, productivity, and developer satisfaction in this domain. In contrast to existing studies, we performed our studies with experienced developers in addition to students. Experienced developers are usually confronted with the development and evolution of much larger projects. Specifically, they need to develop and integrate new software components and to migrate code, tasks that are not covered by these studies. The subjects of our experiments have to implement fully functional Joomla extensions during the stressed development scenarios. To this end, we use the previously presented MDE infrastructure and follow the proposed MDE concept (cf. Section 4.2). However, the study design of our controlled experiment is similar to the ones described in these works, since we are interested in related variables like quality and productivity. Therefore, we compare our findings to the results of these works within the interpretation and lessons learned sections below.

7.2 Quantitative Analysis - Conducting a Controlled Experiment

In this section we present the methodology, procedure, and observations of a profitability evaluation based on a controlled experiment with WCMS extension developers. This experiment has been conducted twice as part of an exemplary pilot evaluation for the Joomla WCMS. By applying the previously described MDE infrastructure during the confirmed development scenarios, we *compare conventional WCMS extension development with MDE*. The goal of this experiment is to quantitatively research the impact of MDE in terms of development speed and code quality. In addition, we investigate whether inexperienced developers can develop WCMS extensions of appropriate quality using an MDE infrastructure.

In accordance to Panach et al. [176], it is impossible to evaluate all pros and cons of MDE within one experiment. As previously described, several works have addressed the evaluation of MDE adoptions in industry and during experimental investigations. The common results highlight the advantages of MDE in terms of quality enhancement, effort reduction, productivity growth, and stakeholder/developer satisfaction. In this context, we aim to extend these results by studying the effect of MDE in the context of WCMS extension development (addressing problem statement 6) in order to answer the following research questions:

RQ2.1: Can MDE affect productivity during Joomla extension development?

RQ2.2: Can MDE affect the software quality of Joomla extensions?

7.2.1 Method

By conducting a controlled experiment based on development scenario 1 and 2 (cf. Section 4.2), we aim to validate the effect of MDE in a systematic, disciplined, controlled and computable manner [177]. To this end, we follow empirical guidelines as presented in [213], [118], and [260] as well as the experimental design of Panach et al. [176] which fits to our desired goal. These guidelines propose the formulation of hypotheses which can be tested in order to answer the addressed research questions. Therefore, we formulate null hypotheses (H0) which indicate no effect based on our selected treatment. The goal of our experiment is to falsify these hypotheses.

We formulate the following null hypotheses which address RQ2.1 and RQ2.2:

H0₁: The developer productivity during Joomla extension development applying MDE is similar to productivity following a traditional development method.

H0₂: The software quality of Joomla extensions developed by applying MDE is similar to software quality following a traditional development method.

Subjects

For the study, we selected 14 developers with significant expertise in Joomla extension development. We justify the selection of student participants with their comparable performance to professionals when using new software development tools [201]. In the conducted survey, the participants stated that they used a variety of commonly used IDEs. All participants stated that they have at least 2-5 years of experience as software developers with a satisfactory to high experience with web technologies. 4 participants have more than 6 years and 2 participants more than 10 years of experience in software development. Only one participant stated that he never implemented software fragments in order to fulfil given requirements. Table F.1 and F.2 in Appendix F summarize the self assessment results with regard to general software development experience.

To ensure sufficient knowledge in extension development for the Joomla WCMS, we conducted a questionnaire and an external knowledge assessment at the beginning of the experiment, based on a multiple-choice test. Table F.3 in Appendix F shows the role of the participants and the time spend with WCMSs in general. Some participants had taken on several roles (user, administrator, developer). Considering the experience in Joomla development, all participants stated that they developed software extensions for this specific WCMS. 5 participants were industrial Joomla extension developers with a high level of experience (2 - 10 years of experience). 9 participants were students from an intensive course on Joomla programming, 5 of them work productively as Joomla developers in a university context. 5 participants stated that they also augmented an existing extension by a new dependent one (scenario 2). In the external assessment, we found that all participants have knowledge in extension development. However, 2 participants showed a knowledge deficit in detailed extension development (MVC interaction).

Additionally, we conducted an assessment addressing the modelling experience of the participants. As the result in Table F.4 in Appendix F shows, all participants had average to high experience in modelling. Based on our external assessment, the modelling experience was higher as the estimated self-perception. Most of the participants had modelling experience with UML.

Moreover, we asked questions in order to get an impression of the open-mindedness towards MDE. To this end, we asked for an estimation of the need for a tool/method for the transformation of requirement documents into models and/or implementations using a five-point likert scale (see Table F.5 in Appendix F). 29% estimated an average need, whereas 50% estimated a high and 21% a very high need for such tools and methods. 71% of the participants used code generators before. This shows that none of the participants was disinclined from the outset.

Variables Definition

Experimentation requires the identification of adequate independent and dependent variables. *Independent variables (factors)* can be controlled in order to investigate their effects on dependent variables. The *dependent variables (response variables)* are the observed subjects of change based on the effects of the treatments. To address the defined RQs mentioned above, each RQ needs to be operationalized in terms of a dependent variable. Our presented experiment studies the effect of two different development methods (levels): Traditional development (control) and MDE (treatment) of WCMS extensions. So, the independent variable is *development method*.

RQ2.1 requires a dependent variable for measuring the effect of MDE to developer *productivity*. Productivity is often measured as amount of fulfilled requirements to effort ratio [176]. In our experiments, we decided for a constant effort (maximum time duration during development sessions). Therefore, we measure productivity in terms of the *amount of fulfilled requirements* based on passed test cases for each requirement (see the checklists in Appendix E).

RQ2.2 necessitates a dependent variable for the effect of the treatments to the *quality* of developed WCMS extensions. We selected a variable tailored to the quality requirements of the WCMS domain: General coding standard for PHP-based web application development. At the time of the experiment two standards were popular in the domain, the Joomla coding standard [166] and the more standardized PSR-2 (now PSR-12) [179]. In our conducted expert interviews (cf. Section 4.1), developers stated that they rather follow official coding standards, such as the latter coding style recommendation. This standard aims to "*reduce cognitive friction when scanning code from different authors*" [87], thus contributing to maintainability, one of the internal quality characteristics in the ISO25010 standard [99]. This standard is also adhered by the Joomla extensions which are developed with the JooMDD infrastructure. To measure the effect of MDE, we asked our participants to adhere to PSR-2 while implementing the requirements. We measured quality in terms of the *amount of code style violations to LoC ratio* for each requirement. We used the *PHP_CodeSniffer* [220] tool for violation detection and *PHPLOC* [22] for measuring lines of code in each implemented view. Both tools are standard tools in the PHP community.

Design

As detailed presented by Juristo and Moreno [118], various and adaptable study designs for controlled experiments exist. The design must fit to the number of subjects and treatments and must adequately address noise factors and learning effects in order to ensure valid and statistically significant results. As described above, we address one factor with two levels and few subjects. Following the common *between-groups design* [43] requires to divide the subjects into two groups whereas each group applies a different treatment for the same problem. However, this design requires a large group of subjects and is more suitable for simple tasks. In contrast the common *within groups (crossover) design* [43, 118] requires a fewer number of subjects, since each group applies the same treatments successively (in a cross-over design). Though, this design is highly vulnerable in terms of learning effect. Therefore, different experimental objects (problems) are required which are addressed by the groups. Adopting this design during the comparison of traditional development and MDE necessitates to let one group start with MDE followed by the traditional method. This may lead to motivation biases due to the expected positive effect of MDE. In [176], the authors compare common experimental designs and discuss their advantages and disadvantages with regard to a similar setting in a more detailed manner. Based on this comparison, we decided to follow an adapted *within groups design (paired design blocked by experimental objects* [118, 176])) with two randomized groups. Both groups started with conventional programming followed by a model-driven development session. To encounter a possible learning effect, we handed out different tasks to the groups. So we address the small number of subjects, possible learning effects, and emerging biases due to the order of the development method. To avoid bias due to one of the tasks being more complicated, we randomized the assignment of tasks to development methodologies between participants [118]. The design is shown in Table 7.1.

Table 7.1: Study Design

Session	Factor	Requirement A	Requirement B
1	Traditional	Group 1	Group 2
2	MDE	Group 2	Group 1

The tasks, based on two different requirements of similar complexity, were handed out during the development sessions. To fulfil the requirements, the groups had to implement extensions for a university management (requirement A) and a custom-relationship management (requirement B). Figure 7.1 illustrates a possible solution model with the entities of requirement B. Group 1 had to implement the first requirement by hand and the second one with MDE, whereas group 2 started with the second requirement followed with the first one. 10 subjects stated that they realised extensions with similar complexity before (cf. Table F.8 in Appendix F).

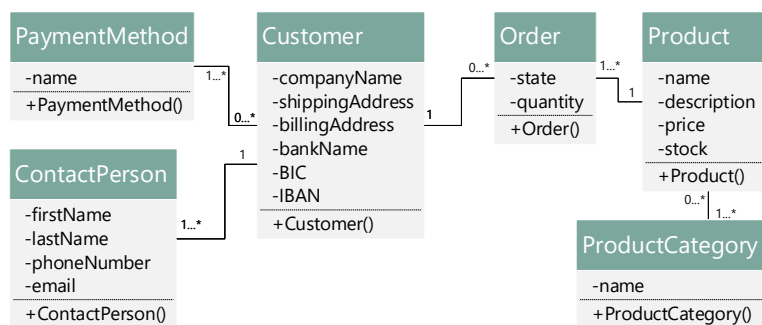


Figure 7.1: Entity Model of Possible Solution for CRM Requirement

Both requirements consisted of an independent Joomla component (scenario 1) with 14 views in total. In particular, 6 list views and 6 edit views for the management of each entity in the administration section (backend), as well as 1 list view and 1 details view for the end-user (frontend). Each view consists of an representation part as well as a CRUD implementation in the backend section. In addition, the subjects were asked to implement a dependent module (scenario 2) which illustrates data of the implemented component. So, in total we had 28 test items for the subsequent evaluation of the results. Appendix E includes the requirements description (Figure E.19 and Figure E.23) and the detailed specification lists for the test items (Figure E.20-E.22 and Figure E.24-E.26).

Instrumentation

During the development sessions the subjects were free to use a development environment of their choice for conventional programming. 12 subjects used a JetBrains IDE (cf. Table F.8 in Appendix F), whereas 2 subjects did not provide information about the IDE they used. In the traditional development session, they were allowed to make use of boilerplate code generators such as the built-in Joomla extension boilerplate generator. 3 subjects stated, that they used existing generators during session 1 (see Table F.9 in Appendix F). During the model-driven development session all subjects had to use the JooMDD web IDE (cf. Section 5.4) for extension development. So, we minimize technical noise regarding the installation of IDE plugins.

For all subjects, we prepared accounts for the web IDE which had to be used during the MDE session. So, we ensured persistence of the created artefacts like models and generated code. Moreover, we prepared a corresponding Joomla testing instance for each subject. So, they were able to test extensions within an isolated environment in addition to local testing.

Statistical analysis

To analyse our results statistically, we tested the two null hypotheses H_{01} and H_{02} using a standard hypothesis testing approach. First, to select an appropriate test, we first check whether our measurement data are normally distributed by creating $Q-Q$ (*quantile-quantile*) plots [232] and applying the Shapiro-Wilk test [211]. Alternatively, the non-parametric *Kolmogorov-Smirnov test* [50] can be applied to test a data set for normal distribution. This test, however, is rather used for larger sample sizes, whereas the Shapiro-Wilk test is more powerful for small sample sizes (cf. [157]).

Our study design is based on two samples with different groups which apply the same development method in a session with different requirements as blocking factor. These may have an effect on the observed variables but their effect is not of interest. Nevertheless, we test the hypothesis for both requirements to address this aspect. Popular candidates for hypothesis testing are ANOVA, t-tests [158, 118]. ANOVA tests are typically used for more than two levels of one or more than one factor designs, whereas t-tests compare the means of two sample sets which are normal distributed. However, t-tests require normally distributed data sets. In contrast, the non-parametric Mann-Whitney U test [92] is applicable for data sets which are not normally distributed. Both tests yields a *p-value*, which allows to reject the null hypothesis in case that p is smaller than an upfront-defined significance threshold. In our tests, we use a standard significance threshold of $\alpha = 0.05$.

Furthermore, we consider the effect sizes for the comparisons. Whereas Cohen's d [49] is commonly used for normally distributed data sets, Vargha and Delaney's A_{12} score [240] measures the effect size when the data is not normally distributed. The latter measures effects on a scale between 0 and 1. Vargha and Delaney suggest to interpret the A_{12} score using the following reference values: $0.56 = \text{small}$; $0.64 = \text{medium}$; $0.71 = \text{large}$ [240].

Procedure

Before the actual development session started, a presentation including an experiment and task description was given. This also included the presentation of the definition-of-done (DoD) specification for the resulting extensions. To ensure anonymous handling of the results and eliminate possible biases between the subjects and the experimenter, the subjects were identified by a random subject-ID, which had to be written down on each artefact they filled out during the experiment. After completing a demographic questionnaire, the subjects had to complete the knowledge assessment (development, Joomla, MDE), followed by the two programming sessions. The knowledge assessment consisted of a self-assessment and external assessment part. During the development sessions, the subjects used their own notebooks with their familiar development set-up of choice.

At the start of the first session, a requirements specification was given to all subjects. Both groups had to implement Joomla extensions (1 independent component and 1 dependent module), whereby group 1 had to consider requirement A, while group 2 had to follow requirement B. In a presentation before the second session, an overview of the eJSL DSL, the code generator, and the web IDE was given. In the second session, the subjects had to implement the remaining requirement in a model-driven manner. In each session, the participants had to check the fulfilled requirements in the specification list.

After each session the subjects had to submit their solutions and answer questions considering the development method as well as the quantity and quality of the development results. At the end, a closing questionnaire was conducted, to get insights to the acceptance of the MDE approach. After 9 hours the experiment ended.

The whole procedure, including the duration of each step, is illustrated in Figure 7.2. The presentation and documents which were handed out to the participants, can be found in Appendix E and as online appendix in [189].

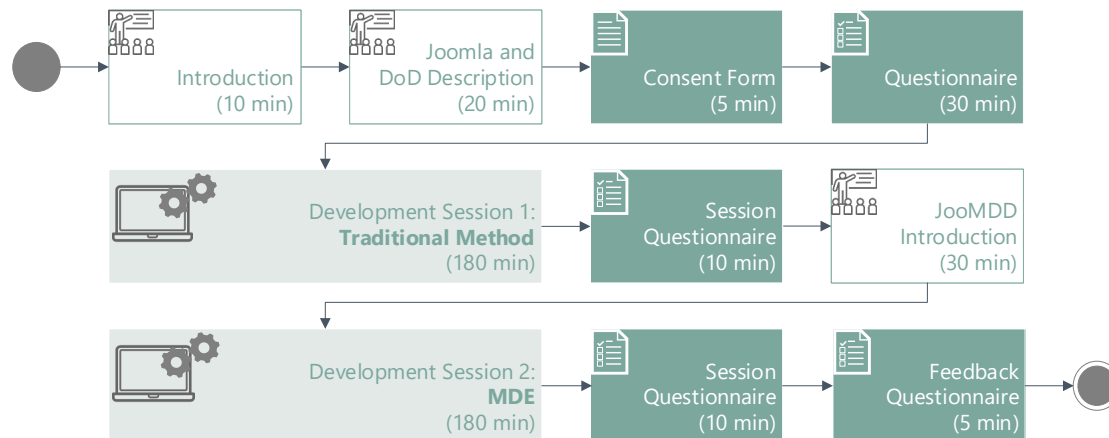


Figure 7.2: Procedure Overview

7.2.2 Results

The test results of the submitted solutions are presented below based on the observed dependent variable (productivity, quality). In addition to overall results, we present a detailed description to gain insight about the effects of MDE on the development of different requirement groups. The row data for each subject is included in Appendix F. In order to verify these results, we test each corresponding null hypothesis based on the previously described statistical analysis.

Productivity

Table 7.2 summarizes the results of the controlled experiment based on the amount of fulfilled requirements per development sessions (3h each). In order to gain general productivity growth insights, we count the amount of passed test cases and build the average percentage of requirement fulfilment. These results are clustered by requirement groups (A and B) with the respective mean and (to expunge outliers) median productivity coefficient as well as the standard deviation (SD) for both development sessions (baseline, MDE). As the table shows, the overall mean coefficient between the baseline session and the session with MDE varies between 5.9 and 11.7 and the overall median coefficient varies between 14.6 and 18.2.

Table 7.2: Productivity Results: Overview (Amount of passed Test Cases)

Requirement		Baseline (%)	MDD (%)	Coefficient
A	<i>mean</i>	7.8	91.3	11.7
	<i>median</i>	6.3	90.9	14.4
	<i>SD</i>	6.9	4.4	
B	<i>mean</i>	9.6	56	5.8
	<i>median</i>	3.6	59.7	16.6
	<i>SD</i>	10.3	34.4	
Overall	<i>mean</i>	8.7	73.7	8.5
	<i>median</i>	4.9	89.2	18.2
	<i>SD</i>	8.5	29.9	

Figure 7.3 illustrates the box plot for the overall productivity result to visualize the productivity differences and variances. The productivity variance of requirement A is quite small for both treatments, whereas MDE of requirement B shows a wide range of productivity amounts. However, this result is based on two outliers (8% and 13%) which were not removed due to the small sample size. Nevertheless, the median of the overall MDE productivity ratio is 30% lower for requirement B. This indicates an effect based on the complexity of the requirement.

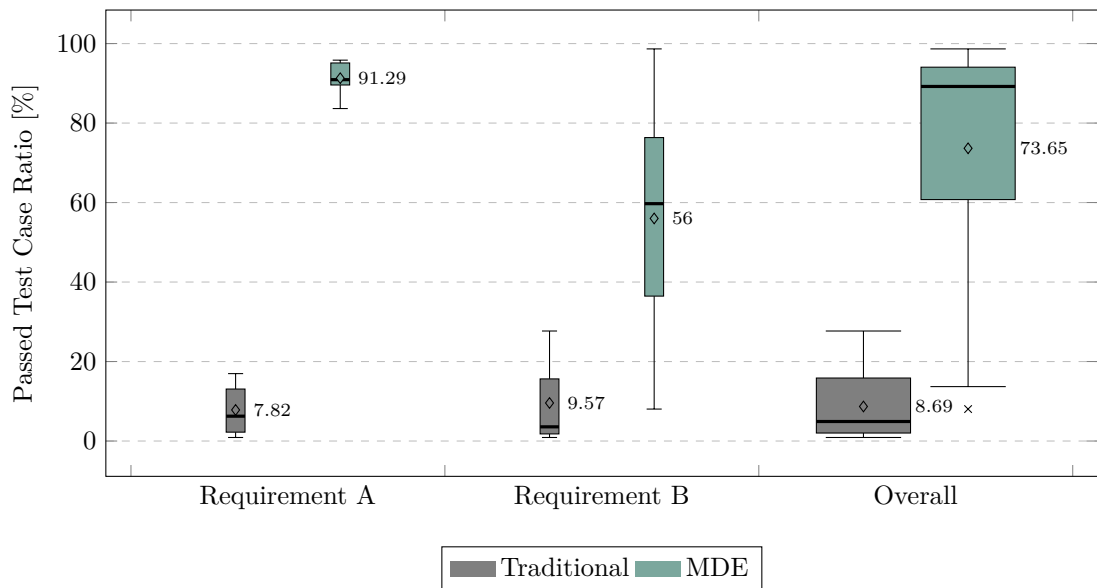


Figure 7.3: Passed Test Case Ratio (Overview)

Detailed Results

Table 7.3 - 7.6 present more detailed results for our productivity measurements. For each subject we measured the percentage of passed tests and summarized the results based on relevant requirement groups in each development session. For each requirement group, we calculated the mean and (to expunge outliers) the median.

In Table 7.3 the overall percentage of functional completeness of the implemented *component structure* by all participants is summarized. This requirement group considers a component that is installable, supports multi-language ability (by language files) and provides update scripts which are processed during a re-installation step.

Table 7.3: Productivity Results: Detailed Insights (Component Structure)

Requirement		Baseline (%)	MDD (%)	Coefficient
A	<i>mean</i>	53.6	100	1.9
	<i>median</i>	50	100	2
	<i>SD</i>	22.5	0	
B	<i>mean</i>	67.9	71.4	1.1
	<i>median</i>	75	100	1.3
	<i>SD</i>	31.3	48.8	
Overall	<i>mean</i>	60.7	85.7	1.4
	<i>median</i>	75	100	1.3
	<i>SD</i>	27.2	36.3	

The following table (Table 7.4) includes the overall percentage of implemented view features (e.g. table columns, filters, orderings, correct fields and HTML field types) of all implemented *component views*.

Table 7.4: Productivity Results: Detailed Insights (Component Views)

Requirement		Baseline (%)	MDD (%)	Coefficient
A	<i>mean</i>	5.7	88	15.4
	<i>median</i>	5.4	90.2	16.7
	<i>SD</i>	6.4	6.3	
B	<i>mean</i>	5.61	55.6	9.9
	<i>median</i>	0	61.3	N/A
	<i>SD</i>	7	35.1	
Overall	<i>mean</i>	5.7	71.8	12.6
	<i>median</i>	2.7	87.7	32.5
	<i>SD</i>	6.5	29.5	

Table 7.5 summarizes the overall percentage of implemented *component CRUD* functionality for each view. This includes the required CRUD buttons and a correct implementation of the associated actions. In Table 7.6 we provide the overall percentage of fulfilled requirements based on a module that is installable, uses the data of the implemented component, and illustrates the data in a module position. The *module* requirement group represents scenario 2 (development of a dependent extension), whereas the previously presented requirement groups in union represent development scenario 1 (development of an independent extension).

Table 7.5: Productivity Results: Detailed Insights (Component CRUD)

Requirement		Baseline (%)	MDD (%)	Coefficient
A	<i>mean</i>	7.1	97.6	13.8
	<i>median</i>	4.2	100	23.8
	<i>SD</i>	7.9	6.3	
B	<i>mean</i>	10.1	57.4	5.7
	<i>median</i>	0	66.7	N/A
	<i>SD</i>	15.6	33	
Overall	<i>mean</i>	8.6	77.5	9
	<i>median</i>	2.1	91.7	43.7
	<i>SD</i>	12	30.9	

Table 7.6: Productivity Results: Detailed Insights (Module)

Requirement		Baseline (%)	MDD (%)	Coefficient
A	<i>mean</i>	0	52.4	N/A
	<i>median</i>	0	66.7	N/A
	<i>SD</i>	0	50.4	
B	<i>mean</i>	0	28.6	N/A
	<i>median</i>	0	0	N/A
	<i>SD</i>	0	48.8	
Overall	<i>mean</i>	0	40.5	N/A
	<i>median</i>	0	0	N/A
	<i>SD</i>	0	49.2	

Hypothesis Testing

The previously presented results show a significant (positive) effect of MDE during Joomla extension development. To verify these results statistically, the corresponding null hypothesis H_{01} (*The developer productivity during Joomla extension development applying MDE is similar to productivity following a traditional development method*) has to be rejected. To elicit an adequate hypothesis test, we have to evaluate, if the data set is normally distributed. To this end, we created Q-Q plots, which are illustrated in Figure 7.4. Based on the Q-Q plots, normal distribution of our productivity data sets can be assumed vaguely.

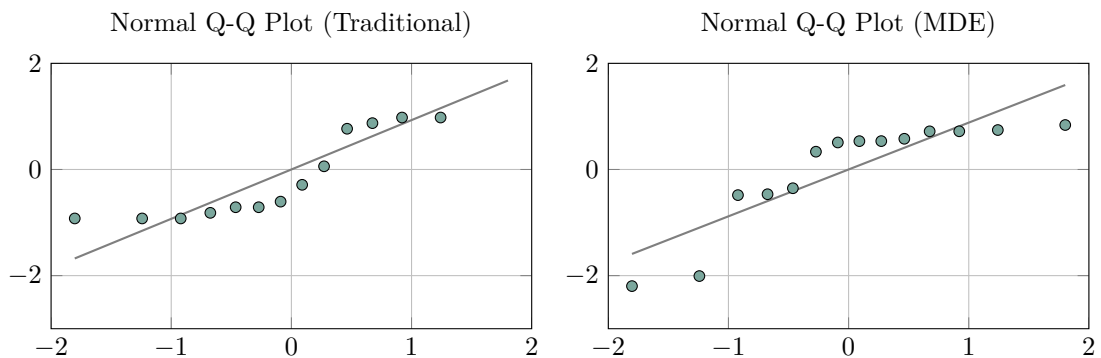


Figure 7.4: Q-Q Plot for Productivity Result Sets (Traditional/MDE)

However, applying the Shapiro-Wilk test [211] on our data sets results in the test statistic results of 0.845 (traditional) and 0.762 (MDD). Based on this test, the null hypothesis can be rejected, if the test statistic is below the critical value of 0.875 for the significance level $\alpha = 0.05$. Since both values are below the critical value, we have to reject the null hypothesis and *assume that our data sets are not normal distributed*. Therefore, we applied a non-parametric Mann–Whitney U test, comparing the mean of two data sets (traditional, MDD). According to this test, there is a statistically significant difference in the productivity results of both treatments with $U = 11$, $Z = -4.318$, and $p = 1.58 \cdot 10^{-5}$. Considering the effect size based on the A_{12} measure yields a score of 0.944. Consequently, we can quantify the effect of using MDD to productivity as *large* [240].

Quality

In Table 7.7, the overall results of the quality measurements are presented. As previously described, we measured the number of code style violations to LoC ratios for all implemented views of session 1. Based on these views we measured the exact same views from session 2 only, even there might be more views. So, we can compare the ratios for the same views considering the low productivity of session 1. In the first session, the subjects implemented 4 different views in total for requirement A, whereas for requirement B they implemented 12 different views. However, the high number for requirement B relates to one subject who used a boilerplate generator, which generated most of the views (cf. Appendix F). As Table 7.7 illustrates, the overall mean coefficient between the baseline and the MDE session varies between 0.42 and 0.47 times less violation and the overall median coefficient varies between 0.42 and 0.43 times less violations with MDE.

Table 7.7: Quality Results: Overview (Violations/LoC)

Requirement		Baseline (%)	MDD (%)	Coefficient
A	<i>mean</i>	1.42	0.66	0.47
	<i>median</i>	1.6	0.67	0.42
	<i>SD</i>	0.48	0.04	
B	<i>mean</i>	1.77	0.75	0.42
	<i>median</i>	1.58	0.67	0.43
	<i>SD</i>	1.58	0.21	
Overall	<i>mean</i>	1.59	0.71	0.44
	<i>median</i>	1.59	0.67	0.42
	<i>SD</i>	1.11	0.15	

To use as many artefacts as possible for our quality assessment, we considered all implemented views from both development sessions, even if they did not pass our test cases. This approach may potentially provide an advantage to traditional development style, for which fewer solutions were handed in for the more complex extension types (cf. the results presented above). To mitigate bias, we present the quality results itemized by different extension types (which avoids bias especially in the simpler cases). Bias in the complex cases is mitigated by the fact that, in the worst case, the bias is negative against our approach – the reported quality benefit presents a lower bound.

In Figure 7.5, the corresponding box plot for the overall result of the violation to LoC ratio is presented. As previously described, one outlier (4.42%) led to a higher mean value for the ratio of requirement B in the first session. However, the median of both requirements is quite similar for the respective development method. This indicates, that the requirement had no explicit effect on the measured code quality.

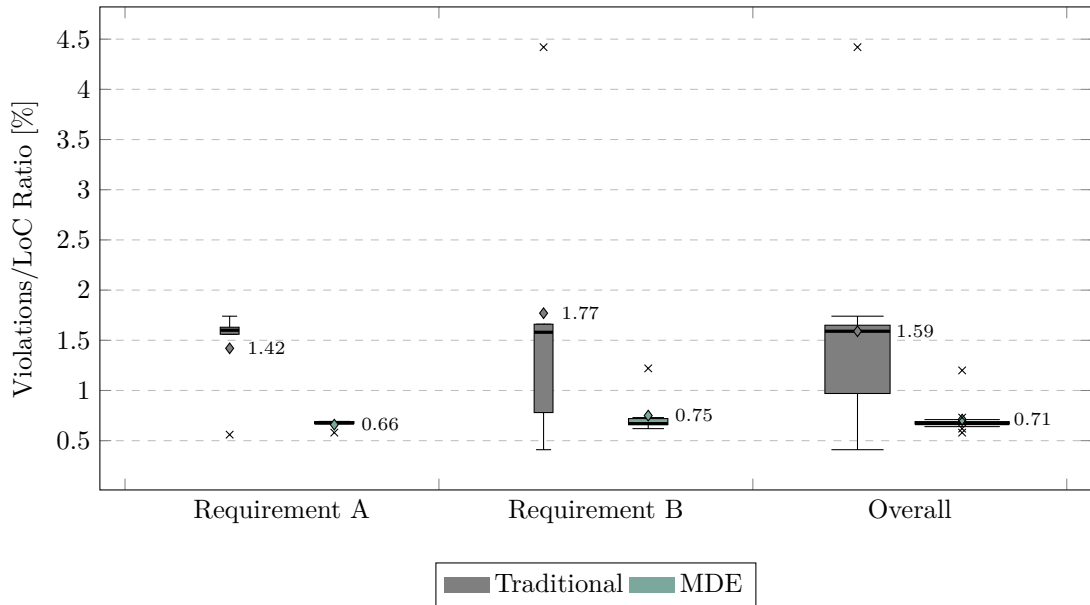


Figure 7.5: Code Style Violations / LoC Ratio (Overview)

Detailed Results

In Table 7.8 - 7.10, we present more detailed results of the code style measurements. Similar to the detailed results of the productivity measurement, we calculated the mean, median, and standard deviation (SD) for each requirement based on the implemented views of session 1. In contrast to the detailed productivity results, we measured the violations/LoC ratio for edit/list and details views of components as well as for implemented modules. The representation and CRUD parts of the views are not explicitly measured, since they are interwoven within the same files.

Table 7.8 shows the overall percentage of code style violations of the implemented *component list views* of session 1. In session 2, the subjects produced 0.38 to 0.43 times less errors for the same views.

Table 7.8: Quality Results: Detailed Insights (Component Views: List)

Requirement		Baseline (%)	MDD (%)	Coefficient
A	<i>mean</i>	1.55	0.60	0.39
	<i>median</i>	1.83	0.54	0.30
	<i>SD</i>	0.68	0.12	
B	<i>mean</i>	1.46	0.63	0.43
	<i>median</i>	1.43	0.54	0.38
	<i>SD</i>	0.74	0.19	
Overall	<i>mean</i>	1.51	0.61	0.41
	<i>median</i>	1.63	0.54	0.33
	<i>SD</i>	0.67	0.15	

In Table 7.9 the overall percentage of code style violations of implemented *component edit views* is summarized. Similar to the list view results, the violations/LoC ratio was reduced in the MDE session (0.31 - 0.56 times less).

Table 7.9: Quality Results: Detailed Insights (Component Views: Edit)

Requirement		Baseline (%)	MDD (%)	Coefficient
A	<i>mean</i>	1.32	0.74	0.56
	<i>median</i>	1.27	0.81	0.64
	<i>SD</i>	0.46	0.12	
B	<i>mean</i>	2.84	0.87	0.31
	<i>median</i>	1.23	0.80	0.65
	<i>SD</i>	4.05	0.26	
Overall	<i>mean</i>	2.08	0.81	0.39
	<i>median</i>	1.25	0.81	0.65
	<i>SD</i>	2.83	0.21	

A comparison of code style violations in modules is not possible, due to the fact that no subject submitted adequate module code after the first session. Nevertheless, we inspected the submitted modules of the MDE session in order to compare the violation/LoC ratio with the ratios of the component views. As Table 7.10 shows, the overall ratio is between 0.38% and 0.4%. This value is lower than the overall ratios of the component views after the MDE session (0.61% and 0.8%).

Table 7.10: Quality Results: Detailed Insights (Module)

Requirement		Baseline (%)	MDD (%)	Coefficient
A	<i>mean</i>	0	0.38	N/A
	<i>median</i>	0	0.38	N/A
	<i>SD</i>	0	0.02	
B	<i>mean</i>	0	0.40	N/A
	<i>median</i>	0	0.40	N/A
	<i>SD</i>	0	0	
Overall	<i>mean</i>	0	0.38	N/A
	<i>median</i>	0	0.39	N/A
	<i>SD</i>	0	0.02	

Hypothesis Testing

Similar to the productivity result, the previously presented results show a positive effect on the quality of Joomla extension by applying MDE. In order to evaluate the corresponding null hypothesis H_0 (*The software quality of Joomla extensions developed by applying MDE is similar to software quality following a traditional development method.*), we determine a suitable hypothesis test by examining the data set for normal distribution. Here, too, the corresponding Q-Q plots, shown in Figure 7.6, do not allow us to draw conclusions about the normal distribution of our quality data sets based on the different development methods.

Based on the Shapiro-Wilk test [211] our data sets results in the test statistic results of 0.564 (traditional) and 0.524 (MDD). Therefore, the null hypothesis that the data is normally distributed can be rejected, since the test statistic is below the critical value of 0.875 for the significance level $\alpha = 0.05$. So, we applied the non-parametric Mann-Whitney U test, which results in quality results of both treatments with $U = 29$, $Z = -2.593$, and $p = 0.01$ which is below α . Based on the A_{12} score of 0.793, we can quantify the effect of using MDD to the quality measurement as large [240].

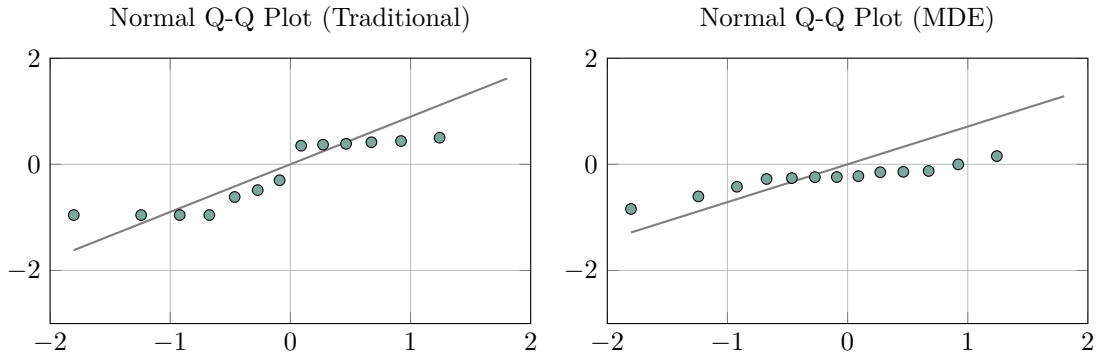


Figure 7.6: Q-Q Plot for Quality Result Sets (Traditional/MDE)

7.2.3 Discussion

Based on the previously presented results we can answer our defined research questions RQ2.1 and RQ2.2. Considering the *effect of MDE on productivity* during WCMS extension development (RQ2.1), we found that even the lowest measured mean coefficient is higher than 5. This shows that the subjects were significantly more productive by applying MDE during the implementation of each requirement in session 2 of the experiment. This is statistically confirmed by the huge effect size of 0.944 (A_{12} score).

Regarding the *effect of MDE on software quality* of WCMS extensions (RQ2.2), the subjects submitted extensions with improved code quality by reducing the amount of code style violations in average at least by the factor 0.42. This result is especially remarkable due to the initial advantage for traditional development, for which less implementations for the more complicated requirements were submitted.

Both experiment results adhere to the interview statements and our previous research, according to which a large amount of extensions consists of generic code for standard views with CRUD functionality. By applying MDE, these extension parts can be developed faster with better quality. This supports the following alternative hypotheses to both rejected null hypotheses:

- H1₁:** The developer productivity during Joomla extension development applying MDE is higher than productivity following a traditional development method.
- H1₂:** The software quality of Joomla extensions developed by applying MDE is higher to software quality following a traditional development method.

The same applies to the development of dependent extensions, whereas the significance of the module development requirement should be interpreted with caution. The experiment design did not allow to conduct a separate module development session. Only one subject decided to implement a module during the first session without any results. The other subjects focused on component development within the time slot of the first session. Due to the fact that all subjects were faster during the second session, more of them were able to develop the required module.

An additional observation based on the experiment results considers the developer satisfaction during the MDE session. Based on the collected feedback after the experiment, all subjects stated that the development with JooMDD was more comfortable. Though, we observed that inexperienced extension developers (<1 year of Joomla experience) were able to fulfil similar amounts of requirements using JooMDD (cf. Table F.10 in Appendix F). Moreover, only one subject refined the generated solution after code generation (cf. Table F.13 in Appendix F). This indicates a positive answer addressing a further research question: *Is it possible for inexperienced developers to develop WCMS extension by using an MDE infrastructure?*

In order to answer this new research question and confirm the previously discussed results, a replication of the experiment with more subjects should be applied. By the instantiation of our study design with further requirements, development scenarios, and MDE infrastructures for other WCMSs, the results may be extended to a broader WCMS context. We propose, to split the development of dependent and independent extensions into separate experiment instantiations to gain better results for the comparison of both treatments. However, based on the complexity of the requirements and the WCMS context, experiment extensions may make it more difficult to find suitable experiment participants, especially from the non-academic area.

7.3 Qualitative Analysis - MDE Workshop with Industrial Practitioners

During the controlled experiment, we focused on forward engineering of the first two scenarios and did not address scenario 3 which require reverse engineering steps (e.g. migration), due to the high effort for understanding and implementing the requirements with two different development methodologies. So, in addition to the quantitative analysis of applying MDE in the WCMS domain as presented in the previous section, we conducted field research with four extension developers of the Joomla community in December 2017. To complement the results with additional qualitative insights regarding the usefulness, acceptance, and open challenges of MDE during our stressed scenarios in this work, we conducted a semi-controlled hands-on tutorial with four extension developers of the Joomla community. We observed the developers adopting MDE during three development sessions including forward engineering and reverse engineering steps. So, we address the following research question:

RQ2.3: Can common WCMS extension development scenarios be addressed with an MDE infrastructure?

This allows conclusions to the applicability of the proposed MDE concepts from Section 4.2 with the MDE infrastructure instantiation as presented in the previous Chapter 6.

7.3.1 Method

Below, we describe the setting of the workshop including a description of the subjects, the design of the tutorial, instrumentation, and procedure.

Subjects

For the workshop, we asked extension developers from the Joomla community to take part in an extension developer meeting. Such meetings are common events in the domain, allowing to exchange ideas. Due to first points of contact with our MDE approach and the JooMDD tool set during Dutch Joomla events, we asked developers from the Dutch Joomla community to instantiate our tutorial as part of one of their developer meetings. Four extension developers with a high level of experience (5-13 years) participated in the workshop. All of them had good knowledge of the processes and problems during extension development and migration.

Design

To provide replication of the workshop based on the addressed development scenarios, we propose a repeatable tutorial design. This includes the specification of suitable requirements which can be handed out to the subjects. Due to the limited time slot of one day, the requirements must be realisable in short time but should be complex enough to allow further conclusions with high external validity. Moreover, adequate solutions must be provided to allow the subjects to proceed with the tutorial, even if they get stuck during the realisation of a requirement.

Our tutorial design is oriented on the three main development scenarios, which we address in this work (cf. Section 2.3 and Section 4.2). Therefore, we ask all subjects to apply *MDE during the development of an independent and dependent extension and the migration of an existing extension package* within three consecutive sessions.

MDE of an Independent Joomla Component (Scenario 1): In the first session, we instantiate the proposed MDE concept for scenario 1 (cf. Figure 4.3 in Section 4.2). Similar to the tasks of the controlled experiment, we asked the subject to implement a Joomla component in the first part of the tutorial, since they are the most commonly developed type of independent extensions. We set the task to develop a conference management component as an extension to the Joomla core (cf. the conference showcase model which is presented in Section 5.5). We decided to keep the complexity of the requirement as low as possible to ensure a high realisation coverage by the subjects. To stipulate the requirements, we specified a class model including the conference management entities. The goal of the first session was to develop a component for the management of conference data by standard views with CRUD functionality, similar to the requirements of the controlled experiment. The resulting component must at least consist of 4 list views and 4 edit views for the management in the backend and 8 views for the frontend representation of the entities. For every view the respective MVC and CRUD code has to be generated as well. Our reference extension model for this scenario has a total of 230 LoC. This includes 4 data entities and 8 different pages which are used for both the frontend and backend. The generated component, including 16 views, consists of 17k LoC. We did not consider more complex extension logic due to the time restriction of the workshop of 6 hours including training (cf. procedure description below).

MDE of a Dependent Joomla Module (Scenario 2): The second session is an instantiation of our proposed MDE concept for scenario 2 (cf. Figure 4.4 in Section 4.2). The task was to add a new Joomla module to the existing conference component developed in the first session. The model should use the MVC model of the component (cf. Section 6.2.2) to provide a new representation of the conference talks within a Joomla site to which the conference component and the module are deployed. Once installed, the module should work together with the already installed conference component (cf. Figure 6.14 in Section 6.2.2). As previously mentioned, we did not expect more complex extension logic in the module due to the time restriction. With more time, this scenario would be realisable, e.g. by developing a dependent component with more complex application logic based on operations on retrieved information from the dependency extension.

Model-Driven Migration of a Legacy Joomla Component from Joomla 3 to Joomla 4 (Scenario 3): In the third session, we required the code migration of an existing Joomla component from Joomla platform version 3 to the Joomla 4 platform which was in an early alpha state in 2017. So, this session represents an instantiation of our suggested MDE concept for scenario 3 (cf. Figure 4.6 in Section 4.2). Even though the new major release of Joomla requires a completely new extension structure (cf. Section 6.2.2), the migrated component should include the same features as the existing extension for the old Joomla version. So, the whole extension structure of an existing Joomla 3 component has to be migrated to the required Joomla 4 structure. Once installed to a Joomla 4 instance, the component views should also be displayed homogeneously and work properly, such as the components for Joomla 3.

Instrumentation

In order to reduce technical noise, we asked the subjects to use the JooMDD web IDE (cf. Section 5.4) during the development sessions. As described above, the web IDE integrates all MDE infrastructure components homogeneously. This includes the eJSL model editor, showcase models, code generator for Joomla 3 and Joomla 4 extensions, and the model extraction tool JExt2eJSL (cf. Chapter 5 and 6). Additionally, we provided a Joomla installation of the latest available version (3.8) at that time (December 2017) to ensure equal conditions for all subjects.

Procedure

We started the tutorial by introducing our MDE infrastructure JooMDD and the JooMDD web IDE. This included a detailed DSL description based on an example model and the presentation of the editor features. The presentation can be found in Appendix G.

The first session started by introducing the requirements for the conference component. In this context, we also proposed a possible development procedure using the JooMDD web IDE. This procedure comprises the use of an example model in the web IDE and changing it to the required conference structure. In the next step, the subjects had to generate a component based on their specified model. As part of the introductory presentation, we explained the structure of the generated code and presented a possible solution. As a next step, the subjects had to install the component to the Joomla installation, which we provided. The subjects then had to check if the extension was installed properly and if it worked as homogeneous part of the website. To this end, they had to create some conference data and test the common CRUD functionality. In addition, they had to create frontend menu entries, to check, if the frontend representation of the conference entities works properly. As next step, the subjects had to refine their existing model iteratively. They had to add a new data entity and pages to display and manage this new entity. Delete operations were not allowed since the JooMDD version at that time did not support an intelligent update mechanism. After the model refinement, the subjects had to re-generate the component and reinstall it to the Joomla installation. After that, the subjects had to check again, if the extension works properly. If everything was done correctly, the existing conference data should be still available in the system.

By handing out the requirements for the dependent module, the second session started. This scenario requires an existing extension package of a conference management component. To this end, the subjects could use the already downloaded extension package from scenario 1 or an extension package which we handed out as possible solution. As next step, they had to upload the extension package to the JooMDD web IDE and use the JExt2eJSL tool for model extraction from the uploaded component package. By using the generated component for model extraction, we make sure that the input extension matches the Joomla standard file and code schemes to ensure that the extracted models are as complete as possible. After this step, the subjects had to inspect the resulting model and make sure that these model elements are annotated with an `@preserve` tag (cf. Section 5.2.1) to avoid the generation of code for them. This tag is used by the JooMDD code generator later to ignore the annotated elements. Additionally, if the resulting model contains some validation errors (e.g. illegal identifiers), the subjects had to refactor these model elements. A further step in this session required the subjects to augment the resulting model by a new Joomla module definition with references to the extracted component-specific model elements. Then, using the new model as input, the subjects had to use the code generator of the web IDE to create an installable extension package of the new module. To complete this session, the developers were asked to install the module to our provided Joomla installation to which the conference component is already deployed (session 1). If it has been installed properly, the subjects had to create a module instance, which has to be placed on the frontend section of the website. If everything worked properly, the module had to illustrate the data of the already installed component similar to Figure 6.14 in Section 6.2.2.

At the beginning of the third session, we presented the requirements for the migration scenario. Based on the required reverse engineering actions, the first steps of the procedure were similar to the ones described for scenario 2. The subjects had to use an installable extension package of a Joomla 3 component, upload it to the web IDE, extract a model, and refactor that model. Again, we decided to use the conference component to ensure a full model extraction. However, in contrast to session 2, the subjects had to remove all `@preserve` annotations in this session. In session 2, the extracted component information was only relevant for the module generation, whereas in this session the whole component had to be migrated. To avoid naming problems

during further steps, the subjects had to add a J4 prefix to the component name in the model. After the model refinement step, the subjects had to generate the component by choosing J4 as generator option in the web IDE and download the resulting extension package. During the tutorial in 2017, the version of the code generator did not generate fully operable components but created the correct new file structure as proposed in 2017 with the main code changes for Joomla 4. This was due to the missing documentation and unstable state of the early alpha release. Therefore, the subjects had to inspect the new components to get an overview of the new component structure.

After 6 hours the tutorial ended. To provide direct feedback of the subjects, we subsequently conducted interviews with the subjects addressing the MDD approach during the scenarios. These interview results were part of the interview set which is presented in Section 4.1 in Chapter 4. In Figure 7.7 the described procedure is illustrated.

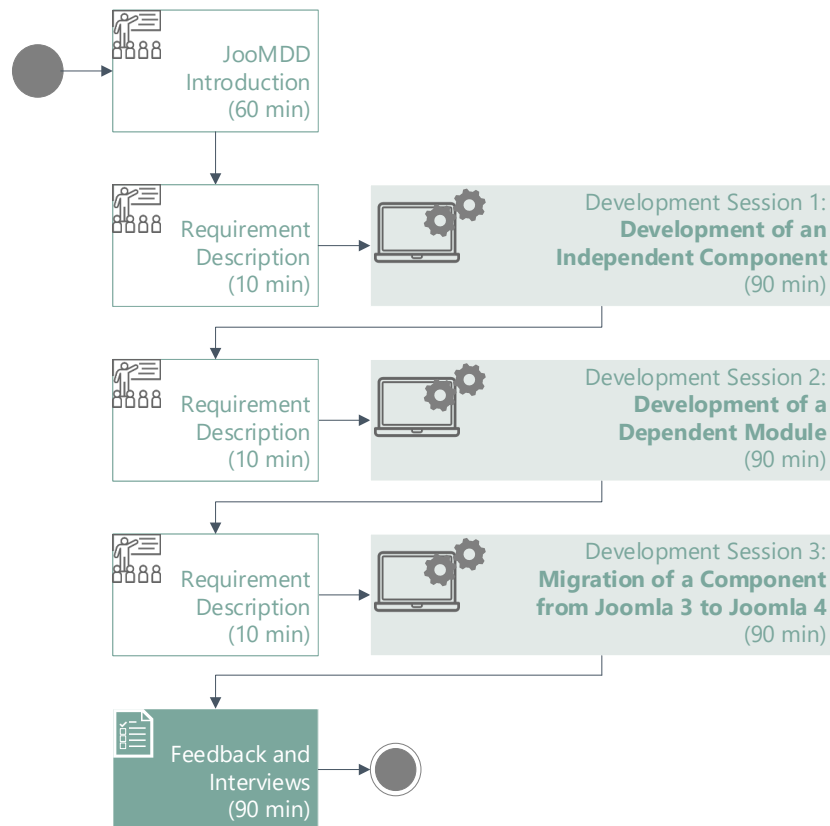


Figure 7.7: Tutorial Procedure Overview

7.3.2 Observations

During the tutorial, we made the following observations based on the actual development session addressing a specific development scenario.

Session 1: MDE of an Independent Joomla Component (Scenario 1): Before the first session started, we observed some reservations against the adoption of a model-driven approach. This also applied to the introduced MDE infrastructure JooMDD. Though, after the first session, the subjects were surprised that the tool worked so well. By using the example models as a reference, the developers were able to quickly learn the concepts of the DSL and how to use the model editor

and generator provided by the JooMDD web IDE. Several editor features were well-received, like the auto completion, error validation, and syntactical sugar, such as curly brackets in the DSL which clarify the structure and model hierarchy.

However, some subjects had problems with keywords of the DSL. Especially, the *page* keyword in the model made some problems. The subjects expected the keyword *view*, since pages in the model represent views in actual components. This is due to the specialised Joomla experience of the subjects. Another technical aversion, we observed, relates the usability of the web IDE. Even though three of the subjects liked the platform-independent editor, the functions of the buttons have not been clear enough without our explanation. One of the subjects disliked the platform-independent solution and would have preferred to use the available IDE plugin for PhpStorm.

After 20 minutes all subjects had installed their first generated component to the provided Joomla installation. We did not observe different results between participant with more or less technological knowledge. The resulting components after the first session fulfilled our requirements, so that all subjects were able to use them as reference within the following sessions.

Session 2: MDE of a Dependent Joomla Module (Scenario 2): During the second session we observed that two of the subjects had problems with the resulting model after the model extraction step, even though they used the generated component from the first session. Since the model was not completely free of validation errors, the developers found it hard to orient themselves, due to the mass of generated model code they are not used to. As described in Section 6.4.2, the JExt2eJSL tool extracts as much information as possible in order to ensure high discovery completeness. This is in contrast to the relatively simple models on which the subjects had to operate in the first session. However, with some help, the subjects were able to create the new modules in minutes and test them, deployed to the provided Joomla installation.

Session 3: Model-Driven Migration of a Legacy Joomla Component from Joomla 3 to Joomla 4 (Scenario 3): In the third session we made the observation that, except for one participant, the group had no experience in extension development for the new Joomla 4 version. However, by using the JooMDD facilities and following the predefined steps as described above, the group was able to create their first Joomla 4 components, based on the previously generated conference component for Joomla 3. Since the process was similar to the one from session 2, the subjects required less help to use the tools correctly. They were fascinated by the applied scenario, since the whole process did not require more than 5 minutes and 4 clicks for the example component. Due to the fact that no migration steps are defined in any documentation, the subjects were grateful to use the generated extension as first reference for their future extension development.

7.3.3 Discussion

The previously presented observations during the tutorial with extension developers from the Dutch Joomla community allow conclusions in order to answer our defined research question RQ2.3. Addressing the first and second development scenario (development of independent and dependent extensions), we observed high acceptance by the subjects, similar to the observations of the controlled experiment. The subjects were able to adopt MDE during these session by using the provided MDE infrastructure JooMDD. In a short time, the subjects were able to produce installable extensions which were operable on a running Joomla installation. However, the complexity of the required conference management is relatively low in comparison to the requirements specified during the controlled experiment. Moreover, no complex application logic was required, what threatens the validity of the results.

Considering scenario 3 (migration of a legacy extension), RQ2.3 can also be answered positively. The subjects were able to migrate an existing extension with the given tools in a model-driven manner. This indicates a positive answer for the development scenario 4 and 5, even though

these scenarios were not explicitly evaluated. The additional scenarios include a similar process including reverse and forward engineering steps in order to discover model information from existing extensions which is then used for code translation. However, to answer the research question in consideration of all scenarios, further executions of the tutorial with an extension of these scenarios should be applied.

Negative remarks during the tutorial exclusively addressed language features of the eJSL DSL or the usability of the JooMDD web IDE. The MDE approach itself was received positively. This leads to the assumption, that MDE is a suitable alternative development method in the WCMS domain. However, to increase the external validity of our conclusion requires to repeat the tutorial with more subjects from other communities, since our discussion is based on the relatively small number of 4 subjects. By using MDE infrastructures for other WCMSs, we can address the research question to a broader WCMS context.

7.4 Lessons Learned

In this section we share the general lessons learned of using an MDE approach within the WCMS domain based on our conducted studies among WCMS extension developers as also presented in [185] and [190]. Besides the results of the presented studies in this chapter, we also incorporate the findings from the expert interviews presented in Section 4.1 and the realised case studies which are presented as part of the generator evaluation in Section 6.5.1, as they are closely related to the findings of this chapter.

Most of the lessons learned are consistent to the ones presented by Whittle et al. in [258]. This applies especially to the following ones:

Finding the right problem is crucial. All stressed development scenarios for WCMS extensions, which are described earlier in this work (cf. 2.3), have proven to be significant. However, the migration scenario as adopted during the tutorial is considered as especially pressing and got most attention.

More focus on processes, not only on tools. Developers ask for supportive wizards supporting them in following pre-defined processes as they occurred in selected application scenarios. This includes dialogues and pre-defined artefacts which can be used out of the box.

Match tools to people, not the other way around. Developers refused working with the Eclipse IDE¹. Instead, they are used to common IDEs in the domain such as the ones provided by JetBrains or web IDEs and await corresponding tool support. In this context, as developers pointed out, MDE has the potential to reduce error susceptibility in contrast to clone-and-own approaches.

Additionally, we found three specific sub-lessons:

1. ***Integrate MDE tooling seamlessly into already used tool environments.*** Developers also asked to consider possibilities for custom code integration into generated code.
2. ***Use domain terminology as much as possible.*** A DSL dialect may better reflect the developer's understanding of a specific domain (such as WCMS extension development with Joomla).
3. ***Handle models as usual development artefacts.*** Developers specifically asked for version management support to consider model histories.

¹For potential reasons see [121].

In addition, we have found further lessons learned which are in accordance to existing MDE adoptions like [31], [234], [23]:

Apply MDE to develop components instead of whole systems. While certain kinds of system components are well suited for MDE adoptions, others may be not. The developers shall be guided to the promising applications.

MDE for learning new platform versions. By automatically migrating a vast part of a WCMS extension, developers can learn how a new platform version (here Joomla 4) shall be used. It also becomes easier to add individual code where needed.

MDE for teaching activities. Teaching a complex system to inexperienced developers can be overwhelming for them, due to technical hurdles (cf. [234]). Using an MDE infrastructure to obfuscate technical details and produce high quality software components encourages students to become acquainted with a new technology. Furthermore, they can do changes on a high abstraction level in the model and re-generate the software artefact on demand.

MDE for rapid prototyping. Since MDE enables rapid development of software artefacts that cover a high amount of domain standards, it can also be used for rapid prototyping, e.g. within an iterative development process (cf. [23]). The prototype, realised by applying MDE, can be successively refined to the final product or even be completely discarded in an early iteration, if it does not fulfil the stakeholders' requirements.

7.5 Threats to Validity

Notwithstanding the promising results of both presented studies, they are subject to a number of validity threats. Below, we discuss *from which threats our studies suffer* and describe *how we avoided typical threats* during experimentation. To this end, we follow the classification which is discussed by Wohlin et al. in [260]. This classification is based on four types of validity threats: *construct*, *internal*, *conclusion*, and *external*.

7.5.1 Construct Validity

To ensure construct validity, we must ensure, that our studies are *close to reality without biases based on inadequate procedures*. In the experiment and tutorial, we study practical applicability of MDE by focusing on three development scenarios that we consider as common in the domain. These scenarios have been confirmed as frequently occurring scenarios by industrial practitioners from the domain (cf. Section 4.1). However, as previously addressed as validity threat in Section 4.4, additional scenarios exist which are not considered yet. Such scenarios must be studied in future work.

All *experiment measurements* were done by the experimenters, even though the subjects had to work through the requirements specification lists for the test items (Figure E.20-E.22 and Figure E.24-E.26). We performed functional tests regarding productivity and code style checks addressing the quality of the submitted results by ourselves. So, the students were not able to falsify the measurement results. However, our measure of quality only focuses on adherence to coding guidelines, which is positively correlated with maintainability [87]. Software quality, though, is a comprehensive construct with further concerns. Based on the ISO/IEC 25010 standard [98], the quality aspect includes several sub-characteristics which can be composed as external and internal quality. External facilitates characteristics like functional suitability, performance efficiency, security, and usability, whereas internal quality addresses the maintainability of a software product. Additional experiments and measurements with the focus on these sub-characteristics are required to extend our findings regarding the effect of MDE on software quality.

In order to address *evaluation apprehension* by the subjects, we only choose volunteer participants from academia and industry during the experiment and only industrial practitioners during the workshop. Additionally, all documents and development results of the experiment were submitted anonymized, to reduce biases based on relationship between the experimenters and the subjects.

7.5.2 Internal Validity

Internal validity is given, if we can assure that a causal relationship between the treatment we used in our experiment and the observed results exists. Based on our decision of using different requirements during the two sessions, we address the threat of *learning effects* during the experiment. Besides affecting the outcome based on the requirements, a learning effect could also affect the choice of the development approach during the traditional development session. This effect is avoided by our study design: traditional in the first session, MDE in the second. Another design would have affected the outcome, as some of the subjects would have used other boilerplate generators, if we had exchanged the treatments between the groups during the sessions. This also concerns *subject motivation* which was guaranteed due to the chosen study design. Otherwise, subjects might find it frustrating to realise the given requirement following a traditional development method after applying MDE. Additionally, our study design avoids the *compensatory rivalry* threat, since all subjects applied the same treatment.

7.5.3 Conclusion Validity

The *reliability of the results* relies on the quality of the artefacts provided to the participants, in particular, the tasks and examples. To mitigate the associated threat, we worked with examples and tasks that are already well-proven from use in teaching and measured the results with objective metrics. A severe threat to conclusion validity is based on the *statistical significance* due to our small sample size. To address this threat in the experiment, we applied the blocked within groups design. So, we could collect data from all subjects during both sessions, which we were able to compare directly. However, the sample size during the tutorial was too small to make any reliable conclusion.

To avoid that a *heterogeneous knowledge of the subjects* affect the outcome, we applied external knowledge assessments at the beginning of the experiment (multiple-choice tests). So, we assured that all subjects provide an adequate background in extension development and modelling, required to apply both development methods.

During the measurements, we used the complete data sets without removing the outliers. So we avoided the threat of *data fishing*, which is often applied to mine data to get a specific result, but may lead to incorrect conclusion. Our results, however, may be distorted due to the outliers we included during measurement. Though, there is no effect on the conclusion validity, since they lead to an advantage for traditional development. Removing them would only strengthen our conclusion.

Our studies suffer from the threat of *random irrelevancies*, since we cannot ensure that all subjects spend the complete time with extension development. However, since we did not explicitly measure time during the studies, there is no direct effect on the conclusions. Again, the significant experiment result would not be different, if we could avoid this threat.

7.5.4 External Validity

By applying our MDE approach within real-world development projects throughout the chosen development scenarios with experts from a specific WCMS domain, we contrast to typical academic research in the same domain. Though, in order to guarantee external validity, we must ensure that the observed results *can be generalised* beyond the scope of the conducted studies.

Since we only considered extension development of the Joomla WCMS as pilot examination, the external validity is threatened. It yet has to be investigated, if MDE as method to fulfil the selected requirements is also suitable for other WCMSs, in particular WordPress, the most popular WCMS nowadays. Though, since Joomla has the most complex extension mechanism, it is likely that the positive results for Joomla may also generalize to other WCMSs like WordPress. However, a new code generator and model extractor is required for the specific needs of each given WCMS.

Another external validity threat is based on *sample size* which is still relatively small due to the fact that we involved experts from the domain as participants. Even though their background ensures reliable quantitative and qualitative results, further studies have to be conducted to allow generalisable conclusions.

Our work suffers from a third external validity threat, since our experiment and tutorial *rely on specific tooling*, namely, our JooMDD infrastructure. We chose JooMDD, since it was the only tool available fully supporting our three considered scenarios. Though, it has to be shown, that the given infrastructure is representative enough to increase external validity of the findings. Moreover, it would be worthwhile to compare the ability of different tools to support developers during a subset of the scenarios.

An additional external validity threat is based on the textual syntax of our proposed DSL, whose design was informed by available examples from the Xtext framework developers, rather than a user study. While variations in the textual syntax could affect the productivity, there are some inherent trade-offs. For example, while our syntax for entity definition could be more concise, we consider the use of keywords such as **Attribute** as beneficial to non-expert users due to their explanatory value.

Moreover, our experimental setting prohibited the use of industrial large-scale example applications including more complex application logic in the requested extensions, due to the required effort for understanding a large-scale system. We argue that the resulting extensions which were developed during the studies (experiment and workshop), are still representative for many applications in the field, even though they mainly support CRUD operations. This assumption is supported by the amount of CRUD implementations in popular WCMS extensions which are published in public extension directories (cf. Section 4.1.1 on Page 53).

*A model-driven engineering approach
is profitable in the domain of WCMSs.*

– Dennis Priefer (et al.)

Conclusively, we summarize the contributions of this work which addresses MDE of WCMS extensions. Additionally, we discuss potential solutions addressing the limitations of this work and outline further research possibilities based on the presented MDE infrastructure and results of empirical research.

8.1 Summary

The use of web content management systems like WordPress, Joomla, or Drupal has established as a popular choice for the creation of a dynamic web application (57% of all websites [252]). By using a WCMS instance, developers can add additional features by implementing installable extension packages. However, developers face various challenges during development of such extensions: *tremendous amounts of required boilerplate code* (e.g. for CRUD functionality), *interdependencies between extensions*, and *frequently occurring architectural changes of the underlying WCMS platform*. These challenges occur during common development scenarios comprising the *initial development and maintenance of independent and dependent extensions* and the *migration of existing extension code to new platforms* including new major versions of the same platform. So, extension development is a time-consuming and complex task, even for experienced extension developers. Based on these challenges, we defined the following problem statements in Chapter 1:

- ⇒ *Problem Statement 1*: Ensuring high quality in WCMS extensions requires tremendous effort due to required coding guidelines and APIs which have to be implemented.
- ⇒ *Problem Statement 2*: The code migration of existing extensions to new platform (versions) requires tedious effort, especially if the number of extensions to migrate rises.
- ⇒ *Problem Statement 3*: The augmentation and re-engineering of (legacy) extensions requires a time-consuming reverse engineering process.
- ⇒ *Problem Statement 4*: The maintenance of dependent extensions is tedious due to the missing dependency management between extensions.
- ⇒ *Problem Statement 5*: Existing tool support does not support iterative extension development, augmentation of existing legacy extensions, or extension migration to new platforms.

A promising approach to address the challenges faced by extension developers is represented by model-driven engineering. Adopting MDE as development practice, allows developers to define software features within reusable models which abstract the technical knowledge of the targeted system. Using these models as input for platform-specific code generators enables a transformation to software extensions for different WCMS platforms. That is, why we propose the use of *MDE during common development scenarios* as efficient alternative to conventional programming in the WCMS domain in order to address the above-mentioned problem statements.

In this work, we introduced an *MDE infrastructure for the development and maintenance of WCMS extensions*. This infrastructure provides a domain-specific modelling language for WCMS extensions generalizing common extension features. To address problem statement 4, the DSL incorporates features for the specification of dependencies between extensions. The presented DSL supports a smart and dummy approach. So, valid models can be defined including only a minimum of extension information which allows a straightforward forward engineering, e.g. as solution for problem statement 1. Additionally, the DSL allows the specification highly detailed extension information. So, we support reverse engineering approaches, lifting existing extension information to model level as required to address problem statements 2 and 3. To support extension developers during modelling actions, we introduced model editors, well-formedness rules, and a set of showcase models.

Additionally, the MDE infrastructure facilitates a set of transformation tools to apply forward and reverse engineering steps. This includes a code generator using model instances of the introduced DSL, an extension extractor for the code extraction of already deployed WCMS extensions, and a model extractor which creates a model instance of the DSL based on an existing extension package. In accordance with the presented DSL, the generator supports a smart and dummy model approach allowing a straightforward development of high quality extensions (solution to problem statement 1). With the presented reverse engineering facilities, we directly address problem statements 2 and 3. The whole presented infrastructure directly addresses problem statement 5, since it complements existing tools providing a solution for iterative WCMS extension development supporting new and legacy extensions.

To ensure adequacy of the provided MDE infrastructure, we followed a structured research methodology, which proved to be effective. First, we conducted semi-structured interviews with industrial practitioners from the WCMS domain to study the representativeness of common development scenarios which we presented beforehand in the background section. Second, we defined and presented a general solution concept for these scenarios including involved roles, process steps, and MDE infrastructure facilities. Third, we specified functional and non-functional requirements for an adequate MDE infrastructure including the expectations of domain experts. With our presented concepts and the resulting requirements we propose a general solution for problem statements 1-4 following a model-driven approach.

The whole infrastructure was realised by following an agile bottom-up development process as proposed in [244]. We discussed related work and presented the general concepts of the DSL and transformation tools. To show the applicability of these concepts, all of the transformation tools were instantiated for the Joomla WCMS which provides the most sophisticated extension mechanism in the domain (based on a comparison which is presented in the background section). Together with the generally usable DSL, these platform-specific tools are incorporated in the JooMDD infrastructure which is publicly available. The proposed DSL and corresponding model editors were validated with respect to popular guidelines for DSLs and the specified requirements based on domain-specific extension features. The same applies to the transformation tools which were evaluated addressing the specified requirements. We presented three case studies to demonstrate the applicability and functional completeness of the code generator and applied five scalability tests which showed that the generation process is not critical regarding scalability ($\mathcal{O}(n^2)$). Moreover, we discussed validity threats based on the followed methodologies during DSL and tool development.

By using the introduced JooMDD infrastructure during *empirical studies with extension developers from the Joomla community*, we researched the suitability and profitability of MDE during the confirmed development scenarios in a quantitative and qualitative manner. So, we can make a statement about whether an MDE adoption in the WCMS domain, following our general MDE concepts and using our MDE infrastructure, represents a reasonable solution for the collected problem statements.

First, we shared the method, results and conclusions of a controlled experiment which was conducted with 14 developers from academia and industry. We compared conventional extension development with MDE using JooMDD. During the experiment, we focused on the first two scenarios (development of dependent and independent extensions). The results showed a clear gain in productivity and quality by using an MDE infrastructure. We found a productivity growth up to factor 11.4 and a quality increase up to factor 2.4 during the MDE session. These results were verified by applying hypothesis tests (Mann–Whitney U) for the corresponding null hypotheses. Based on the test results, we rejected the null hypotheses and showed, that MDE has a large positive effect on the developer productivity and extension quality based on Vargha and Delaney’s A_{12} score. Moreover, we presented the design and observations of a semi-controlled tutorial with four experienced developers who had to apply the JooMDD infrastructure during the three major development scenarios (development of both dependent and independent extensions and migration of an extension to a new platform version). The goal of this study was to obtain direct qualitative feedback about acceptance, usefulness, and open challenges of our MDE approach. Conclusively, we shared the lessons learned and discussed the threats to validity of the conducted studies.

In conclusion, we answered the two main research questions of this work. We showed how MDE can support developers during common WCMS extension development scenarios (RQ1), and researched to which extend the MDE approach supports developers in terms of development speed during WCMS extensions development and the quality of WCMS extensions (RQ2). The positive outcome of the conducted studies substantiate that MDE is a promising development method in the WCMS domain in contrast to conventional development. So, this work represents a step forward in order to confirm the theory that MDE is profitable in the WCMS domain.

8.2 Outlook

We propose to address the following *limitations and open challenges considering the proposed MDE Infrastructure* (cf. Section 5.6 and Section 6.5) in further work:

Extensions to the Domain-Specific Modelling Language: The proposed DSL for WCMS extensions incorporates the features and requirements which were emphasized by domain experts. Though, it is currently mainly tailored to data-oriented extensions. So, mainly CRUD features are supported, whereas complex features such as references to specific core features cannot be specified by the proposed DSL. The same applies to sophisticated UI specifications, e.g. for component views, which are currently not provided by the presented DSL. We propose an extension of the DSL by means of further language elements to address these limitations. Moreover, the provided core support should be further extended in order to allow more interdependencies of an developed extension with the underlying system. This requires a rigorous investigation of interdependencies of WCMS extensions and underlying core platforms beyond the context of Joomla to ensure a general use of the DSL. In this context we also suggest the provision of additional DSL features for the specification of specialised extension features, e.g. payment features for web shop realisations.

WCMS-Specific Modelling Languages: To allow modellers to specify more technically-oriented features tailored to a specific WCMS, we suggest the introduction of platform-specific sub-languages or dialects which can be used together with the proposed DSL. So, domain experts with less technical knowledge can produce abstract extension models which incorporate their ideas of the extension features, whereas developers with experienced knowledge of a specific WCMS can extend these models by platform-specific design decisions which are typically outsourced to code generators. This allows more individual extension development and will enable expert developers to use their common terminology. In this context, a model-to-model transformation between instances of platform-independent and platform-specific models may increase model reusability.

Tool Implementations for other WCMSs: In order to ensure external validity of our findings, we propose the implementation of generators and RE facilities for additional WCMSs. This allows conclusions regarding the universal applicability of the introduced DSL and the presented transformation tool concepts. As next step, we propose the realisation for WordPress and Drupal, due to their similar extension mechanisms and popularity in the domain. As already described in Section 6.5.2, the integration of the project of Cabot [37, 36] which proposes a simple DSL and corresponding code generator for the WordPress WCMS into our MDE infrastructure seems to be a promising step as part of future work. According to first investigations, all language features of the proposed DSL are covered by our introduced DSL. The same applies to the code generator concepts which show similarities to our concepts.

Custom Code Support: Similar to the most MDE infrastructures, the manual extensibility of generated extension code is limited, due to the provided custom code strategy based on generated dummy code. Since our current infrastructure is tailored to the development of CRUD code, which typically makes up the largest part of an extension, the integration of a sophisticated custom code integration had low priority in this work. However, developers emphasized this feature as a substantial requirement to an MDE infrastructure. Therefore, we suggest to put more attention on this feature as part of further work. A promising approach could be the integration of custom code models which can be referenced by model instances of the proposed DSL.

Due to the existing threats to validity of the empirical studies cf. Section 7.5, we also suggest *further empirical investigation of MDE adoptions in the WCMS domain:*

Validation of the Controlled Experiment Results: To increase the validity of our experiment results, we recommend to repeat the experiment with more subjects. To ensure reliability of the results, the appropriateness of the given requirements during the experiment should be confirmed by domain experts. Moreover, we like to conduct more qualitative studies of extension development for other WCMSs, like WordPress or Drupal. This, however, requires code generators and model extractors for these WCMSs.

Extension of the Study Designs: Since we exclusively studied the effect of MDE on developer productivity and software quality, we propose an extension of the study design of the experiment in order to measure additional dependent variables. Especially the quality aspect could be extended to measure potential effects of MDE on other quality characteristics such as efficiency, compatibility, security, or usability. Moreover, further studies should explicitly research, if it is possible for inexperienced developers to develop WCMS extensions by using an MDE infrastructure. Even though we made first observations in this context, an extension of our study design by suitable definitions of dependent variables should be done as part of future work. The design of the tutorial could be extended by additional requirements and dependent variables in order to allow its adoption as field experiment. So, the study can be applied more systematically to gain empirical evidence of how our MDE infrastructure is used by developers of the domain.

Evaluation of Sub-Scenarios: Since our conducted studies exclusively address development scenarios 1-3, we recommend to conduct additional studies to research the appropriateness of the additional scenarios 4 (partial augmentation of existing extensions with custom features) and scenario 5 (reengineering of a legacy extension in the context of quality assurance). Since our proposed MDE infrastructure allows the application of these scenarios in the Joomla context, we propose to start with suitable case studies with developers from this community.

Identification of Further Scenarios: Further development scenarios, common in the WCMS domain, must be studied. As pointed out by one of the practitioners during our conducted interviews, the abstraction of shared functionality into libraries is a viable development scenario. Moreover, further combinations or sub-scenarios of the described scenarios may exist, such as the presented scenarios 4 and 5. In this context, a further investigation of the applicability of round-trip support should be applied. This enables the consideration of additional scenarios,

e.g. considering dependencies to third-party extensions which are further developed by external developers. The identification of new scenarios requires the definition of new MDE concepts and may necessitate the specification of additional requirements for the MDE infrastructure.

Industrial Application of the MDE Infrastructure: To research the usefulness of our proposed MDE infrastructure, additional case studies in the domain should be conducted. Especially the most pressing migration scenario could be researched in the field, due to upcoming major releases such as the new Joomla 4 version which is announced for being released in 2020. Based on anecdotal evidence, there is interest in the Joomla community in using our MDE infrastructure instantiation JooMDD for the migration of existing extensions. This allows us to apply our infrastructure during actual case examples to infer the usefulness of our approach empirically. Moreover, we can iteratively refine our MDE infrastructure based on new emerging requirements which occur during the realisation of these case examples.

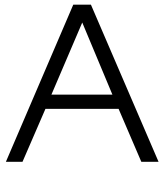
Moreover, we emphasize possible *research directions* to which we discovered points of contact during the work on this thesis. This includes existing contributions which were not addressed in the context of this work.

Development of Model Interpreters: The rising popularity of web frameworks for front-end development such as Angular [79], React [63], and Vue [267] should be considered during further work. The fact that these frameworks are also used in the WCMS domain, such as the Vue framework for the media manager extension in Joomla 4 (cf. [110]), emphasizes their relevance. In order to provide support for such frameworks, we propose to extend the presented MDE infrastructure by model interpreters. While our generator parses model instances of our DSL at compile time and generates installable extensions from them, an interpreter can represent changes to the model live at runtime without having to be recompiled. This is especially suitable for the mentioned front-end development frameworks, which swap most of the logic to the client side of the application. By providing such an interpreter, model refinements can be directly applied in the running application without the need of a compilation step. So, the approach can be used beyond the scope of installable WCMS extensions. However, this strategy requires the implementation of model interpreters, which can be incorporated to the respective application. As part of a student project, an Angular interpreter was realised, which can be used as part of an Angular app which interprets model instances of our eJSL DSL [143]. The examination of this strategy, including the identification of suitable case examples, could be a further research direction. Another benefit of this approach is that we can extend the general usability of the language which goes beyond the scope of Joomla.

Evaluation of MDE Applicability of Existing Code: One aspect that has led to a decline in the popularity of model-driven approaches over the last decade is the inappropriate selection of projects. If the software to develop is too individual, MDE might not be a suitable approach. The same applies to small software projects which can also be adequately addressed by conventional development methods. If a software artefact has to be developed only once without the need of replication as variant of the artefact, MDE might also be too expensive. Developers must consider these aspects when they start a project or think about reengineering of legacy software. This also applies to the WCMS domain, where extensions of various complexity and individuality are developed and maintained. To support developers during the decision of adopting MDE as appropriate development method, we propose an evaluation step as part of the reverse engineering of existing extensions. To this end, we already incorporated an evaluation step to our model discovery concept in Section 6.4. Moreover, we already specified a metric called *RevEngA*. This metric gives a hint about the reverse engineering applicability of an input extension package. To measure the amount of schematically recurring code fragments of a legacy extension, we developed a code clone clustering tool which produces a report of the amount of same files with similar code structures. This tool is used within an instantiation of the concept for Joomla extension packages together with the parsers presented in Section 6.4.2 and is part

of the JExt2eJSL tool of the JooMDD infrastructure. Using this evaluation feature supports Joomla extension developers to evaluate the appropriateness of existing extensions to be further developed with our proposed JooMDD infrastructure.

Automatic Extraction of MDE Infrastructures: One argument against MDE is the pain of initial DSL and tool development for new domains. The same applied to our presented infrastructure, which took several years to develop. Even though meta tools for MDE infrastructure development like the Eclipse Modeling Framework (EMF), Meta Programming System (MPS), or Xtext address this challenge, an appropriate mechanism for the semi-automated creation of infrastructure artefacts are not available. Therefore, an additional research direction could address the reduction of initial effort during MDE infrastructure development. A promising approach is based on the extraction of meta models or grammars and corresponding code templates from existing reference applications. This could be achieved by mining schematically repetitive and generic code fragments using clone detection techniques. The discovered code clusters should be reviewed by technology experts in order to ensure a proper subsequent (automatic) processing. Especially in the WCMS domain, where a tremendous amount of existing extensions exist, this approach could be useful. In the context of this work, the approach could be a supportive strategy to extract code templates for the creation of platform-specific code generators for additional systems like WordPress or Drupal. For the realisation of the approach as tool environment, existing parsers like the ones presented in this work (cf. Section 6.4.2) could be integrated. For the evaluation of resulting tool environments, we propose the use of existing MDE infrastructure instantiations, such as the one introduced in this work, as reference environments. A first publication in this context is made by Rost [196].



Semi-Structured Expert Interview

In the following sections, the original transcribed results of the conducted semi-structured expert interviews are presented. Seven of the Interviews were recorded, whereas one was documented in writing.

Interviewee 1

Interviewer: *Wolf Rost*, Date: *09.12.2017*, Location: *Dongen, NL*

Q: First, if we publish this interview, we will anonymize the data. So we just don't put names in there. Did you develop Joomla extensions?

I1: Not really. I maintained some.

Q: OK you maintained.

I1: I maintained some that's already in place but I also had to move one extension from 1.5 to 2.5. And extension was used on 200 (or so) sites.

Q: How long did it take you to migrate the extension to the new Joomla version?

I1: I don't know it's been some time ago. I think, it was not as hard as I thought it would be. So migrating to the next version took maybe a couple days.

Q: 8 hours per day?

I1: Yes. It was a component that existed of very little. Most stuff first online in the plugins. And I figured it would be a lot of work. And I was lucky it wasn't. In the year later or a year and a half later I found one bug as a result of the migration in all those two hundred sites. Yeah it was funny because it's between two versions of Joomla some names can change so where in the old version it was `com_User` the new version is `com_Users` but we had to use the name of the new one We hadn't seen there was a problem at first.

Q: Can you tell me more about the extensions you migrated? How many? What type of extensions?

I1: This was a specific case. I was working at a company that's doing automation for sports communities in the Netherlands. And one way of using this was publishing data from the sports communities on Joomla sites. So it means that if the community is playing football or hockey or volleyball or something similar, they can have Joomla sites with all the competition data. And the members can log in with their credentials and all those kind of stuff. And the plugin was managing all the communication between their Application, that was a Java application of Oracle and all kind of stuff and the Joomla site which is P.H.P. and on a different server.

Q: Did you ever try to use parts of a third party extension for your own extension? So, like we saw at one of the use cases, did you use the model of a third-party extension to get data for your own extension?

I1: No, it would be a nice case, but not one I use. But it's a use case!

Q: Did you ever experience that the Joomla conventions like the file structure or the naming lead to errors, during the migration for example? You forgot to change the name in one controller or sth. like that and that broke the site.

I1: Yeah maybe. There are always mistakes.

Q: Did it occur a few times or often.

I1: No, I would not say more than regular. I mean it certainly you make mistakes but as long as you don't make the same mistake very often, it's fine

Q: So you could say it occurs like every other error you did not just "Oh, it's this error again. I forgot to rename the class"?

I1: No.

Q: So, you saw our generator approach. Do you have experience with other Joomla generators like the boilerplate generator?

I1: I've played with the Component Creator, but what the component creator does is take away a lot of typing. Which is fine, but what it does not do is, it's difficult to incorporate it in the workflow. So, if you use Component Creator to start a component it's fine and if this component is just implemented and that's it, great. But if you want to take these components and maintain it or to fill it up further or give it more possibilities or functionality, I think the component creator is not really helping. It's only the first bit. But my idea was that if it's only helping me in the first bits of typing a lot of stuff to save a lot of times I can also automate this otherwise. There's no need for a component creator if this is all it's really helping with. Sort of boilerplate would have been fine as well but it wasn't there.

Q: So, you would say that for initializing or for the first step in creating an extension it's helpful but for further development, it's not?

I1: At least not as I've seen it. Maybe it's better now I don't know. There's change in the component creator. Improving also. But yeah.

Q: Which functionality do you expect from a generator tool?

I1: I expect that the generator is not a generator once and change never option. I expect that it's meant to be part of a continuous developing situation.

Q: So not just for the first step.

I1: If it appears to be a bug in my component after six months and I want to be able to go back to the last one that was generated or the last one before that or something like that. And with the components I have seen for generating this was not possible. In the component creator it's like you create something and if you change it afterwards and creating a new one that's in place of the old one. The old one is gone.

Q: You would like to have sth. like a versioning system?

I1: Yeah or at least have a history yes. Because often if you find a bug then it can be a totally new bug. It also can be a known bug that you had before that has returned for some reason.

Q: Last question, Did you augment an existing extension by a new view or module?

I1: Only modules.

Q: So you had a component and you created a new module for it?

I1: Yes.

Q: What was your procedure to do that?

I1: Copy an old one and change what needed to be changed.

Q: How long did it take you? Can you quantify it?

I1: Not much, a couple of hours.

Q: OK. Thank you very much.

Interviewee 2

Interviewer: *Wolf Rost*, Date: *09.12.2017*, Location: *Dongen, NL*

Q: If we publish the results then we don't put any personal data in it.

I2: Ok, it's fine. If you want to put my name somewhere I don't mind. You can always use a link to our website.

Q: First question is, did you develop extensions?

I2: Yes, I have to develop extensions.

Q: For how Long?

I2: The company where working for twelve years and at first we started with only custom templates and I think now the last five years we are also developing extensions.

Q: What kind of extensions?

-
- I2:** Well in the beginning mostly modules and plugins just to make small changes to websites or put some content that you actually need from the database and the last two years we're actually making our own custom components.
- Q:** What size does the component have? How many views? Do you think they are complex?
- I2:** Yeah, they're complex. While we have one really large one we're working on. It's called Image Manager and it has well actually it's only a backend component just to organize the images and it has only one view. So, it has some extra models that so they popping up with some extra information but it's only actually one view.
- Q:** Okay, but there is really much behind it?
- I2:** Yeah, just to make it's really simple for end users to use the extension. That is actually our main goal just to make it as easy as possible for the end user to use the extension. But in the back end there happens a lot of stuff.
- Q:** So, you would say there is a high level of individuality in your component compared to standard list views.
- I2:** Yes.
- Q:** What is your standard procedure to implement components? Do you look at the documentation? Use a reference application?
- I2:** Yes. We also have now our own boilerplate we use just to create a component and we use Grunt for that. It puts all your old files in the right places and creates all the default classes and all the meta information. This is the main way we go and sometimes well if it's just a small component we actually do it by hand but most of the time we use the Grunt installer. And for the modules and plugins I also use a website. It's a really nice one. You put in some information you press a button and you get a standard module with all the information in it.
- Q:** Do you also create extensions, modules, that use data from a component?
- I2:** Yes, we now have a component with five views and also a module and also a couple of plugins. It's a music database where teachers can collect songs they can actually do with the children in the class. So, they open the player, in the player there are all the songs and they can click on the songs and the player will start playing and they have all the song texts next to the player. So that's a really complex application with a lot of fun of use and also a module which is also a player and a lot of plugins who are actually doing all kinds of stuff in the backend.
- Q:** And you used the boilerplate generator for all this?
- I2:** Yes.
- Q:** And does it support other extension types besides components?
- I2:** No, only components. For modules, we use the simple module creator. Since plugins are mostly only four files (it's a plugin, an XML and two language files) that is something I mostly do by hand.
- Q:** Can you quantify how long it takes you to create a component?
- I2:** When we use our boilerplate it still takes us, I think, two hours to set up everything. So, two hours for a component.
- Q:** Without any individual changes. Just to get an installable component.
- I2:** Yes, and to create all the views and make it installable as an empty component. Because every view has different information we have to create the database after creating the component. We have to create all the tables and because there's something that's not in our boilerplate. So, that's what is taking most of the time. Just creating the database and the tables and then also creating the views and queries, sometimes even by hand because that squares quicker. Just log in and creating all the columns. So we actually don't have an actual workflow we use when we make an extension.
- Q:** Did you ever try to use a model from a third party extension for your own extensions?
- I2:** We have worked with the Component Creator and it's I think it's really nice if you're not that good in programming. But if you look at the code, it's not as easy to change all the things like the way it should work. I think in some situations it could be a really nice solution but if you want to make your own components it's better to start from scratch. That's our experience at least.
- Q:** Did you migrate an extension from Joomla version to another?

- I2:** Well, I think the components we've created now are all for Joomla 3 and most of the modules they actually worked on 2 and 3. So we didn't any of component migrations. Probably our first migration will be to Joomla 4 and I think our programmer is already up to the standards of Joomla 4 so Image Manager should also run now on Joomla 4.
- Q:** Did you experience that the Joomla conventions like the file structure or the naming of the classes lead to errors? Are there errors, that occur often?
- I2:** I don't know because I'm not into component creation myself. I'm just mostly testing every view and the code so I'm not writing that much so it is something I have to ask our programmer.
- Q:** In which scenario do you think code generators could help you?
- I2:** Sometimes you need a boilerplate to create everything. It could actually help and as I've seen some examples today (JooMDD tutorial) I think this is really useful to speed up the process and actually when you have to create a standard component which has to do something really easy you can make one really quick. So it's can be a time saver. Yes I'm sure it can.
- Q:** Which functionality do you expect from such a generator tool?
- I2:** I have to see what's missing right now. I was talking about the wizards which can even speed up to the process even more so that's something you have to look into because I think that could be really powerful just a simple wizard that you have to just type in and it will make the whole model for you and I think that just could speed up the process even more.
- Q:** A QA game?
- I2:** Yeah, just fill in the name of the component and then do you want a module created and a plug in or just a component and you have to take which one you need so and the model knows exactly what kind of code to generate. That could be a bigger time timesaver.
- Q:** OK, that's it. Thank you very much.

Interviewee 3

Interviewer: *Wolf Rost*, Date: *09.12.2017*, Location: *Dongen, NL*

- Q:** If we publish this interview with all the other interviews we will make any personal data anonymous. I hope it's OK for you?
- I3:** Oh yeah I've never been to one I think I have nothing to hide I don't care. Whatever you want to know it's just okay with me.
- Q:** Did you develop some extensions?
- I3:** Yeah. Components not that many. Not even a handful I think. Mostly, if I needed I created a plugin. I use a lot of plugins and I do really weird things with plugins. I then think when they're finished "Maybe this would have been something to create a component for" actually sometimes.
- Q:** So, your plugins are quite complex or big?
- I3:** Some. I created quite a specific plugin. A payment plugin. The risk is there is a special Joomla component called formToContent which is a very specialized CCK (Content Construction Kit). This one is really simple. The only thing you can do with it, is create articles. Nothing more, nothing less. Very simple. And there was a guy in Australia and I don't know why but he approached me to create a payment facility for people who would post an article they had to pay before the article was being posted and I created a program to make that work with this CCK. And that was a bit of a complex plugin.
- Q:** So, you also have a high level of individuality in your code? Youn don't just use standard Joomla code?
- I3:** I try to use the Joomla API wherever I can. As much as possible.
- Q:** What is your procedure to create a new plugin? Do you look at the documentation? Do you copy an existing plugin?
- I3:** Well because I have a few now, I copy the ones from myself. Yeah. The first one I created I think I created this from scratch but it's been some time so I can't really remember. But these days I copy a plugin. I created one yesterday. There's another client of mine or a colleague who has a client who wanted a request form he pushed into a cloud-based C.R.M. client relation management. So they have an interface, a web service, and I created a form with RSForm. It's a form component, I think the most used form component for Joomla. And it's easy to interface on a very simple level. So I had it almost working and then I found

out that I couldn't get it completely working so I crafted a small plugin because they have a lot of plugin events scattered all over the code and I could do it like this so.

Q: How long did it take you to create the plugin?

I3: Two hours I think in this case.

Q: So, this was a smaller one?

I3: It was because I had the basic code already. I created the business logic, so to speak, completely separate from Joomla. It was just talking to the API and everything. And then I just had to integrate it. So, the plugin is a wrapper around the actual application logic.

Q: Did you ever try to use parts of a third-party extension in your own extensions. E.g. use the model from another extension to access its data in your plugins?

I3: Yes, I tried that. Actually, I tried to get my hands dirty on a REST component, like so many people because we missed it in Joomla. And I wanted to make it quite general. So, I wanted to really create clean interfaces so it would be either easy to interface with components directly or easy to sort of plug in your own specific component interfaces. And then you run into all kinds of problems. It's amazing how tightly coupled everything is in Joomla.

Q: So that was difficult?

I3: Yeah, you know it's one of those projects I mentioned earlier today that's lying around somewhere unfinished and because it would take too much time for. In Joomla it's very difficult to create because the whole architecture is all too connected it's still too dependent on each other. I mean there are so many funny things you must do to instantiate a model. It's weird.

Q: Did you ever have to migrate from one version to another?

I3: Well, not really I mean I have good contact with Roland. I hear things when he has to migrate to get through to new versions so it's difficult. Well you know it's already difficult to upgrade a website to a new version. Even a simple website with a few extensions from a third party. I still have a few laying around from clients which I didn't think. There is one with a component which I made actually. Was one of the components I did. But it's either very simple or it's so perfectly developed. It didn't need any upgrading it just still works in (version) 3.8. And I don't think I will have the client still when Joomla 4 is coming out so I won't have to upgrade it I think.

Q: Did you ever experience that the Joomla conventions lead to errors in your own extensions. E.g. if you forgot the file structure or naming conventions. Would you say this lead to errors often?

I3: I never experienced it.

Q: Maybe, because of the plugins?

I3: Right. They are quite small quite self-contained.

Q: We presented our generator approach. Did you use some other similar approaches or generators?

I3: For one component once I used Component Creator. It was a simple component. It created well short of also boilerplate and I added the specific client wishes to it. I think in the very beginning I wrote a few shell scripts to generate boilerplate for components and stuff but that's like ten or fifteen years ago. I don't think that's really generated. I mean it's a generator, not much more, very simple.

Q: Which functionality do you expect from such a code generator? What scenarios do you think are suitable?

I3: I think the number of scenarios is limited. So what I've seen from JooMDD (use cases shown in the workshop). That a bit like as much as I expect from it. I think and what I said before the focus should be on what you do should be good and enable to generator to hook into with yourself with your own custom code or with template kind of things that you're not restricted to just the generated part and you're not forced to hack into the generated code but to just have enough possibilities to do at your own stuff. I think there should be the focus and not try to cover too much. Because then I think it will never be finished and it will never be stable because you will run into all kinds of weird bugs and stuff and you'll be working eighty percent of the time on twenty percent of the functionality.

Q: That's it. Thank you very much.

I3: You're welcome.

Interviewee 4

Interviewer: *Wolf Rost*, Date: *09.12.2017*, Location: *Dongen, NL*

Q: If we publish the results of the interviews we will don't put any personal data in it.

I4: You can do what you want. I really have no preference.

Q: Did you develop extensions?

I4: I did and I still do as well.

Q: Since which Joomla version?

I4: I started on mambo. Pre Joomla. So you can basically say since Joomla 1.0.

Q: What kind of extensions do you develop (extension kinds)?

I4: I do all of them. Well, all extensions for Joomla. It's components, modules, plugins. I do not do templates, because templates is a completely different discipline. And that's really the four major ones that are used in Joomla anyway.

Q: Is it focused on components and modules?

I4: No, it's focused on componens but because of the components plugins and modules are part of it to be able to display certain information for modules then and for plugins to catch triggers that are sent by other components and by own components.

Q: What do you think, are they complex in meaning of how many views they have or references between the views in your components?

I4: I think my one component, CSV Improved, is a pretty complex one because if you look at for example there's a view there called template in which you put all your settings you only use for import or export. And depending on some settings you have it's pulling different XML files and rendering them individually. That's is not something I really see in being done in Joomla. So in that sense, I think that view specifically is very complex. Most views are pretty standard because Joomla is based on listing pages and edit pages. But with the template view there's actually an extra toolbar button to switch between the basic mode and advanced mode. You have test buttons for testing FTP connections, testing file locations, and other things like that. Well, that's not something you have in your general edit screen. Because most edit screens are simply for data input and this is doing more than that.

Q: Can you quantify, how many views are standard views and how many views are individual?

I4: I think that ninety percent at least are standard listing views. If I look at for example the logs view which has a listing of all the log files have been recorded in the list field there, after each log, there's a button to show, to download, or to open the log file. That is something that is not standard in Joomla. I actually have it in both my components because both are doing log files and it's been simply an easy way for users to see a log file. But the code behind the billing is standard Joomla again because it simply shows the modal popup and inside it loads the text file.

Q: What is your procedure to implement a new component? Do you look at the documentation or do you copy an existing one?

I4: If I need to start a new component I now would start with the boilerplate that I have created for Joomla. Simply because the basics are there. Of course before I even started I have a good idea of what I want to build. Because I need to see if a component is actually the right approach for what the goal is to be achieved. But if the goal results in having a component to be built, I would use a boilerplate to get started because in PhpStorm it is just a click new project to the project -> component and the whole thing is there and I can get started with. I just need to change. Perhaps the name of the default field because the default field is basically your dashboard. And then you add the other views for whatever functionality you need.

Q: How long did it take you to develop a new component?

I4: Of course, it completely depends on how many views you need and what it needs to do. The boilerplate of course as soon as they have to check out of that it's installable but in general if I look at having a backend view and a frontend view it's between four to eight hours and that of course is still quite a bit more work because in an hour I can make a frontend and backend view but that's a very basic.

Q: What is your procedure if you have an existing component and add a new view?

I4: Copy&paste. Copy an existing view, copy the existing model, change the names, change the database queries where needed. The most work is still in the template file because you have to change the column names

and stuff like that. You have to create the filter if there's a filter. But the basic fastest way is copy&paste because you just have to change a few things in most of the files and it's working. Well and don't forget you have to copy the table files because it's active record procedure so Joomla is always looking for a table file.

- Q:** What is your experience with the Joomla conventions. Do you think the conventions like file structure and naming often lead to errors or did you never experienced that. E.g. if you change a class name.
- I4:** The Joomla convention I think it's fine. Because in all the twelve years Joomla existed, it hasn't been changed. So whenever you learn or learning to work with a new system you just have to learn whatever everybody else is using. It can be good it can be bad but at least it's been consistent over the last twelve years. If I forgot to change a class name that's my bad. The only thing that I think should change is that class names in the frontend and the backend have the same name. Because not only is in IDE it cannot find the class name you want to use. You always have to look "am I in the frontend or am I in the backend and that's the only thing that I would really change. Or maybe not even half a distinction between frontend and backend for your models and your controllers but only for your views.
- Q:** But do you think such errors occurs often?
- I4:** Yes, renaming classes after copy&pasting a model is standard procedure almost. Because if you do one view most likely enough forgetting because you only have two or three files to change but if you are doing a couple of views I would say it's like a hundred percent chance that you forget something somewhere. Because you have to change the name of the model you have to change the name of the controller the name of the view, change the name of the table file. And the other thing you forget is change in the query the table that needs to be queried. That is something also forget. It's a tedious procedure. It would be nice if I do copy paste it would ask me what's the new name of your view and then because the names are always consistent it ask me the name and could consistently change it for me.
- Q:** So that could be a functionality that you expect from a code generator or a generator approach?
- I4:** Yes, that would be amazing. If I now look at JooMDD what we did today, if I add a new view in the model file, I would still copy&paste a view from there because the structures mostly going to be the same. But there's less to change because there's only one single file where I need to change maybe two names or three names then the rest will be generated. So, it's less error prone then what we're doing now.
- Q:** Did you ever try to use parts of a third-party extension for your own extension. For example, do you use a model from a third-party extension to get data for your own extension?
- I4:** I have done that. I'm still doing it. The example here's in my extension I'm exporting prices from a webshop and the whole logic of price calculation - I don't want to recover that - so I'm using the model from the third-party component that has the logic in it and that wasn't easy. Mostly because the way the classes are built. Because in PHP you have public, protected, and private properties and as soon as a property is protected or private I cannot longer access it and that was my main problem. So I was able to create my own class name with exactly the same name as they have. And because they check if the class is already loaded, it would load mine first. And I am extending their class but I need it. Because I'm talking to my class I can actually access the protected methods and that solve my problem. Because they don't have a public facing API I could use and that was, of course, be the most ideal solution. But there's nothing public facing in that sense.
- Q:** Did you ever migrate an extension from one version to another Joomla version.
- I4:** I have been in there since Joomla 1.0 so I had to migrate my extension through all versions.
- Q:** And was was your experience, how long did it take?
- I4:** You should take a year. I think so if you want to do it well. Of course, it taking a year is not that I'm working full time for one year to migrate the extension because the migration happens next to the other work I need to do. The daily business goes on. Maybe without any interruptions, it still will take a couple of months. Of course, it depends on how big the code base is. If you have a component with two views you could be done in a day. But this one with the very complex views a lot of complex logic in the importing and exporting and because I support ten to fifteen other extensions, it needs a lot of testing and work and that just takes a lot of time.
- Q:** Which kind of functionality do you expect from a generator tool?
- I4:** What I would expect is that if I have my logic inside the code generator it would spit out a component in the new style that I put in a different engine and the engine gives me different code to be doing with Joomla 3, Joomla 4, or whatever platform it's supposed to be running on.
- Q:** So it should support generation for different Joomla versions (platforms)?
- I4:** Exactly, because otherwise, I don't need a code generator. I can just change everything manually again. But having the main part of my logic in the code generator would allow me to just export that.

Q: Thank you very much.

Interviewee 5

Interviewer: *Dennis Priefer*, Date: *24.04.2018*, Location: *Gießen/Munich, Germany*,

Note: *The interview was conducted via Skype.*

Q: Ist es ok für dich wenn wir das Interview aufnehmen? Wenn wir es veröffentlichen werden wir es auch anonymisieren.

I5: Ja

Q: Hast du mal grob überschlagen wieviel Erweiterungen du schon entwickelt hast?

I5: Ich denke so 40-50 Komponenten. Zig Plugins und Module. Insgesamt ca. 100.

Q: Für welche Versionen hast du den das gemacht? Mit welcher Version hast du angefangen?

I5: Ich habe schon mit 1.0 angefangen. Komplett durch.

Q: Auch schon mit Joomla4?

I5: Mit einer Joomla-Core-Komponente.

Q: Wie komplex waren deine entwickelten Erweiterungen? Wieviele Views? Wieviele waren davon Standard-CRUD-Views im Backend und Frontend? Hatten du auch komplexe Abhängigkeiten zwischen den Views? Oder waren das eher sehr individuelle Geschichten?

I5: Meine größte Komponenten war eine Mitglieder-Verwaltung. Im Backend hatte die, 33 Views und im Frontend 7-8 Views. Diese Komponente hatte ich dann aber mal in mehrere Komponenten aufgeteilt und im Endeffekt die über 5,6,7 Komponenten, wo immer so 5-6 Views drin war, verteilt. Die dann aber auch mit einander verbunden waren. Eine Komponente hat erkannt ob ein Plugin oder eine andere Komponente installiert war und dann entsprechend verschiedene Features oder Menüpunkte angeboten. Also ganz konkret, eine Mitgliederverwaltung hat halt Mitglieder verwaltet und wenn ich dann ne Locationverwaltung oder eine Kartenverwaltung installiert habe, dann hat die Kartenverwaltung plötzlich, Tabs bekommen mit denen man Mitglieder auf Karten anzeigen lassen kann. Da gabs dann Abhängigkeiten.

Q: Es waren unterschiedliche Kombinationen möglich.

I5: Genau entweder hat es, es dann angezeigt oder nicht.

Q: Hattes du da auch Standard-Crud-Views drin, wie man sie auch aus dem Joomla-Core kennt oder waren das individuelle Geschichten?

I5: Es waren im Backend zu mindest von der Oberfläche her komplett Standard. Also immer Listen mit Form-View. Es waren dann eher die Details die sehr individuell wurden. Wie z.b. das man eigene Formfelder braucht oder Funktionalitäten wie einen Import-Knopf oder irgendwas. Aber es hat immer auf dem Joomla-Core aufgebaut. Es gab immer Controller, Model und View. Wenn etwas gespeichert werden musste, dann wurde tatsächlich immer das Joomla-Speichern verwendet. Und wenn es gepublished werden muss, das Joomla-Publishen und nicht irgendwie was komisches draufgebaut.

Q: Wie gehst du beim entwickeln einer neuen Erweiterung vor?

I5: Also der Vorteil von der Art wie ich es bau ist dass die Komponenten gleich sind, Details unterscheiden sich aber meistens ist es so, dass ich irgend eine Komponente nehme und die kopieren und die Namen anpasse. Also ich habe immer so Basis-View, ein Dashboard ist immer dabei und das nehm ich dann alles packs hin, pass die Namen an und dann immer copy&paste, ja, es ist halt alles gleich. Wenns eine neue List-Views ist, dann wird halt von irgendwo eine List-View kopiert, Name angepasst, Formularfelder angepasst.

Q: Bist da auch schonmal irgendwo auf Probleme gestoßen?

I5: Nein, bis jetzt nicht. Es kann natürlich sein das ein Tippfehler drin ist, aber ich weiß dann immer genau wo der Fehler ist.

Q: Hast du schon einmal einen Codegenerator, wie ComponentCreator, ExtensionBuilder verwendet?

I5: Ich habe sie getestet. Ich habe sie durchgeteste. Also anderst rum. Ich habe irgendwann mal mit Laravel was gebaut und die haben ja so einen Generator drin und im Prinzip finde ich die Idee recht schick. Das man es so zusammenbaut und hab auch die Sachen die es für Joomla gibt angeschaut. Das Problem ist natürlich immer die sind nie auf dem neuesten Stand. Das halte ich für das größte Problem an den Generatoren, also ich finde die Idee schick aber nutze sie selber aus dem Grund nicht, das wenn ich einen Generator benutze

will ich eben so eine eierlegende Wollmilchsau. Wo ich nicht mehr hinterher durchschauen muss ob den die neusten Änderungen von vor zwei Wochen drin sind. Das hindert mich dran. Durch das kopieren wo ich dann auch durchschauen muss, verliere ich nicht viel mehr Zeit.

- Q:** Ok, wie lange brauchst du den wenn du mit dem Clone-and-Own Ansatz so eine Erweiterung erstellst? Im Schnitt, wenn wir sagen ich brauche von dir eine Standard-Komponente mit 5 Views, also Dashboard und 4 weitere Views/Entitäten die ich verwalten will? Was meinst du wie lange du da brauchst? Installierbare Version.
- 15:** Ich schätze mal 2-4 Stunden. Es kommt auf die Komplexität der Felder drauf an. Für eine komplexe Komponente, 10 Jahre. :)
- Q:** Yeah, its a never ending process. Wie gehst du vor wenn du eine neue View einbauen willst?
- 15:** Kommt drauf an was für eine View, aber meistens wird eine bestehende View kopiert, der Name angepasst und entsprechen die Funktionalität hinzugefügt.
- Q:** Mit einer View ist es ja meistens nicht getan, es gehört auch noch ein Model hinzu.
- 15:** Ja, genau, das wird natürlich mitkopiert.
- Q:** Controller, Forms?
- 15:** Ja
- Q:** Also im Prinzip das gleiche. Du kopierst etwas vorhandenes und änderst es ab. Wenn es eine View sein soll die neue Entitäten darstellen soll, wie gehst du da vor auch im Bezug auf die Datenbank?
- 15:** Also im Endeffekt bleibt es bei dem Prozess. Es wird kopiert und rausgeworfen was man nicht braucht und das eingefügt was man braucht.
- Q:** Hast du in deinen Erweiterungen, Referenzen auf existierenden 3rd-Party-Extension?
- 15:** Ja, durch Plugins. Falls die 3rd-Party-Extension und das Plugin installiert ist, wird ein zusätzliches Feld angezeigt, wenn nicht dann nicht. Es ist eine lose Abhängigkeit.
- Q:** Hast du deine Erweiterungen schon migriert?
- 15:** Meine Mitgliederverwaltung habe ich schon das vierte mal neugeschrieben. Einfach aus dem Grund weil ich es hasse wenn da noch altes Zeug drinne ist, das nicht "State of the art" ist. Aber ich glaub seit 1.5 oder so wenn man sich da an die Struktur gehalten hat, hätte ich das nicht machen müssen. Nur nen bisschen Klassen anpassen und so polieren aber nichts neu migrieren oder so. Also von meinen ganzen Erweiterungen habe ich ca 15-20 migriert und den größten Teil irgendwann sterben lassen. Wenn auch diese migriert hätte, hätte ich es auch gescheit gemacht und dafür hatte ich keine Zeit.
- Q:** Wenn es da aber jetzt ein Tool gegeben hätte, dass das für dich gemacht hätte? Hättest du dann mehr migriert?
- 15:** Also wenn das Tool dann wirklich auch "State of the art" und aktuell gewesen wäre hätte ich das sicher gemacht. Ich bin eigentlich voll Fan von solchen Tools, wenn sie nicht veraltet sind. Mein Kriterium ist immer, es muss die neusten Standards unterstützen, dann nutze ich so etwas auch. Aber das tun die wenigsten bis keiner.
- Q:** Wie lange hast du für eine Migration gebraucht?
- 15:** Das ist schwer zu sagen. Bei der Erweiterung mit den 33 Views war ich bestimmt ein 3/4 Jahr dran, 10h-15h pro Woche. Die nächste Migration war dann eigentlich der Schritt wo ich diese Extension auf mehrere Extensions aufgeteilt habe und diese neu aufgebaut und die Gelegenheit genutzt habe um alten Code rauszuwerfen. Bei großen Änderungen habe ich es immer neugeschrieben.
- Q:** Interessant, anderen habe immer einfach migriert.
- 15:** Ja, ich weiß, deshalb gibt es auch soviel Schrott weils keiner macht. Das ist halt immer ein riesen Aufwand.
- Q:** Hast du auch schonmal etwas von einem anderen System oder zu einem anderen System migriert?
- 15:** Nein, ich habe aber schon von Wordpress-Seiten auf Joomla migiert, das waren aber die Daten.
- Q:** Bei was ist die Entwicklung von Erweiterungen in Joomla schlecht?
- 15:** Das viele Klassen z.b. die Google-API entfernt wurde. Das wurde einfach gemacht. Ich habe diese benutzt und muss es nun selbst machen.
- Q:** Du tendierst eigentlich schon dazu das man sich an den Joomla-Standard hält aber dann muss es auch so einfach wie möglich nutzbar sein?

15: Genau

Q: Es gibt ja verschieden Use cases im daily business eines Entwicklers, z.b. das initiale, das weiter entwickeln, etwas vorhandenes um ein z.b. modul erweitern und migration. Siehst du das auch so oder fallen dir noch weitere ein?

15: Ich weiß nicht ob das schon abgedeckt ist mit einem der user cases. Konkretes Beispiel: bau irgendetwas, du hast eine Standardstruktur: Table-Klasse, View, Controller, Model und jetzt kommt die Anforderung: Ich brauch ein neues Feld. Ich habe ein System in dem schon tausende Daten vorhanden sind und jetzt brauch ich eine neues Feld in der Datenbank. Wo im schlimmsten Fall noch ein Fremdschlüssel auf irgendetwas anderes ist. Bei so etwas wäre es nett irgendetwelche unterstützung zu bekommen.

Q: Ok, dieser Anwendungsfall ist auf jedenfall abgedeckt und das wäre etwas was du dir als Entwickler wünsche würdest.

15: Genau, auch für Initial, du startest ein Projekt. Also ich finde den Laravel-Ansatz echt schick. Du baust dir so zu sagen eine Grundstruktur zusammen, sagst ok hier, ich habe das und das und das, habe diese Tabellenfelder und am Schluss will ich es ausgeben, am besten auf Joomla gemappt. Das ich sagen kann ok ich habe eine Listenansicht und ich kann sagen auch dieser Tabelle die Daten nehmen. Ich brauch die Access-Verwaltung von Joomla und das und das und das. Für Standards wie Formulare und Listen müssten man eigentlich gar keinen Code mehr schreiben.

Q: Das sind also so deine Hauptanforderungen an ein Generatortool.

15: Die Hauptanforderung ist dass es Aktuell ist. Damit steht und fällt es und dann soll es soviel Code schreiben wie möglich abnehmen. Und diese Aktualität soll nicht nur initial geben sein sonder es muss auch vorwärtstkompatibel sein. Das also auf Knopfdruck meine Komponente aktualisiert wird. Im Sinne von Framework-Sachen.

Q: Also wenn ich es nochmal wiederhole. Dir ist es sehr wichtig das es aktuell ist, also nur aktueller und qualitativ hochwertiger Code ausgespuckt wird. Zum anderen aber auch die Möglichkeit relativ einfach das ganze zu konfigurieren oder zu modellieren.

15: Da zu möchte ich etwas sagen. Idealerweise nicht, ich bin zum Beispiel kein Composer-Fan. Die Sache, ich bau mir da irgendwie eine JSON- oder XML-Datei zusammen und führe diese dann aus. Das finde ich so...puh...das sagt mir garnicht zu. Laravel ist echt optimal, ich komme da hin und ok, was will ich jetzt machen. Mit Befehlen, bau mir View x,y,z und dann werden ich gefragt. Einen Wizard so zu sagen. In irgend einer Form einen Wizard.

Q: Du wünschst dir also einen Dialog und nicht irgendwo eine Datei.

15: Genau, natürlich was schön wäre wenn man etwas daraus generieren könnte, für das nächste Projekt. Das man also das vorherige Projekt als Basis fürs zweite Projekt nutzen kann. Also dann aber wieder ein Wizard. Ich importiere mir diese Konfigurationsdatei und mit dem Wizard wieder anpassen.

Q: So ein bisschen wie npm -init

15: Genau

Q: Würdest du dir eher eine textuelle Form für diese Konfigurationsdatei wünschen oder auch eine visuelle Editoren im Sinne von Klassendiagramme?

15: Nein, ne, es geht auch wieder um Geschwindigkeit, wenn ich da mit Drag&Drop rumhantieren und so...das ist nicht schnell. Ein Kriterium muss sein, ich muss schneller sein als wenn ich Copy&Paste mache. Für Beginner ist das vielleicht sehr nett aber für mich wäre das nichts. Ich kopiere mir halt so eine Komponente innerhalb 5 Minuten zusammen und dann muss ich noch die Sachen anpassen.

Q: Fällt dir sonst noch etwas ein, was du dir von einem Generatortool wünschen würdest?

15: Datenbank-Versionierung, im Sinne von nicht nur von der Entwicklungsseite her sondern es wäre natürlich auch krass wenn es auch Content-Versionierung wäre. Zum Beispiel verschiedene System zusammenführen mit Content. Das ist noch ein ungelöstest Problem in der Informatik.

Q: Vielen Dank.

Interviewee 6

Interviewer: *Dennis Priefer*, Date: *23.04.2018*, Location: *Gießen, Germany*,

Note: *The Interview was not recorded but documented in writing.*

-
- Q:** Is it ok that I record the interview? If we publish the results, all personal data will be made anonymous.
- I6:** Yes.
- Q:** Did you develop one or more Joomla extensions in the past?
- I6:** Yes.
- Q:** For which Joomla version(s)?
- I6:** Since Mambo.
- Q:** What kind of extension did you develop? (Component, Module, Plugins, or other)
- I6:** Modules and Plugins.
- Q:** How complex was/were the extension(s)? How many views? How many CRUD views? References between extensions? How was the level of individuality?
- I6:** Between 80 and 90 % standard code. Modding of components (overrides). Extended project forks.
- Q:** What is your procedure to implement a new module?
- I6:** Clone-and-own approach. No Boilerplate generators
- Q:** Did you implement dependencies between extensions (component <-> module)?
- I6:** Yes, module dependencies to existing components.
- Q:** How long did it take to develop the extension(s)?
- I6:** Half a day.
- Q:** What is your procedure to augment a component by a template override?
- I6:** Copy and adapt the layout file which has to be overwritten.
- Q:** Did you ever try to use parts of a third party component in your own extension?
- I6:** Yes.
- Q:** Did you migrate an extension (code) from one Joomla version to another?
- I6:** Yes.
- Q:** Can you describe your procedure to migrate extensions?
- I6:** First, we create a Joomla test instance on a test server. Typically, we use a backup of an actual complex Joomla installation as reference and update it to the new version. Then we install the old extension, test it and refine it iteratively until it works. Since we mainly use third-party extensions, we wait for adequate releases and install them. In the last step, we add the template extension and refine it to the new needs.
- Q:** How long do you need for a migration?
- I6:** Time window of 2 years.
- Q:** Did you migrate an extension (code) from one system to another (e.g. WordPress to Joomla)?
- I6:** No.
- Q:** How often did.../Did you ever experience that the Joomla conventions (file structure/naming) lead to errors in your extensions?
- I6:** Sometimes. The typical problems/bugs (view not found, bugs in layout files, incorrect database query).
- Q:** In which scenario could a generator approach be helpful?
- I6:** It could be useful during the initial development and migration step. Also during the augmentation of existing components if existing data is not required. Or if an extension has to be newly developed/reengineered.
- Q:** Which functionality/features do you expect from a model-driven/generator tool?
- I6:** No unnecessary features. Textual specification of features. Consideration of references (extending). Development of modules for existing components. A wizard would be good but there is no urgent need.

Interviewee 7

Interviewer: *Dennis Priefer*, Date: *27.04.2018*, Location: *Gießen, Germany*

- Q:** Hallo P7, schön dass du heute Zeit gefunden hast für das Interview. Ich nehme das Interview jetzt auf und frage dich auch ob das ok ist. Ist das ok?
- I7:** Ja.
- Q:** Alles klar, dann legen wir doch direkt los. Du bist ja auch schon ein erfahrener Joomla-Entwickler. Seit wann bist du denn schon dabei?
- I7:** Das sind jetzt schon 4 Jahre.
- Q:** 4 Jahre, aber du hast mit Joomla 3 angefangen?
- I7:** Genau mit der Migration von Joomla 2.5 auf 3.
- Q:** Ok, das heißt du hast schon mit Erweiterungen gearbeitet, Erweiterungen entwickelt und auch Erweiterungen migriert? Ist das richtig?
- I7:** Eigene Entwicklung nicht, also Weiterentwicklung von Joomla-Komponenten, Plugins und Module und die Migration halt.
- Q:** Die Komponenten, die entwickelt hast, wie komplex würdest du die den beschreiben.
- I7:** THM-Groups ist schon sehr komplex. Im Backend sind es 15 Views. Frontend weniger, ich glaub 7 Views.
- Q:** Würdest du sagen, dass der Code bzw. die Views in der Komponente Standard-Views sind oder sind das eher sehr individuelle Views? Mit Standard-Views meine ich, typische CRUD-Views. Wo man eine Listenansicht hat mit den Entitäten, man kann die Entitäten dann bearbeiten. Man klickt drauf dann öffnet sich eine Detailansicht und man bearbeitet die, kann speichern und dann werden sie in der Liste wieder dargestellt. Das ist eine Standard-View, wie man sie auch im Core-Erweiterungen, wie Users etc.. Würdest du sagen das THM-Groups größtenteils aus Standard-Views oder aus individuellen Views besteht?
- I7:** Backend ist auf jeden Fall, größtenteils Standard-Views, also Liste-Views mit CRUD-Funktionalitäten. Frontend, z.b. Profil-Edit ist sehr individuell.
- Q:** Im Frontend gibt es aber auch Listenansichten und wenn man draufklickt bekommt man Editieransichten. Würdest du das nicht als Standard-View sehen?
- I7:** Ich bin jetzt eher von der Advanced-View ausgegangen.
- Q:** Könntest du das Quantifizieren? Also, du sagst im Backend, hauptsächlich Standard-Views. Wieviel Prozent würdest du denn sagen der Komponenten an sich, würdest du sagen entspricht einer Standardansicht und wieviel sind individuell? Entspricht also keiner Joomla-Komponente.
- I7:** 60-65
- Q:** Ok, das deckt sich mit unseren Erfahrungen. Hat die Komponente Abhängigkeiten zu anderen Erweiterungen?
- I7:** Ja, zum System-Plugin, Content-Modul, Profile-Modul.
- Q:** Also es gibt Abhängigkeiten die von der Komponenten selbst abhängen. Hat den die Komponente selbst Abhängigkeiten zu anderen Erweiterungen?
- I7:** Früher bestand eine Abhängigkeit zu einer Bibliothek. Aber das ist jetzt alles in die Komponenten verlagert worden.
- Q:** Hat die Komponente Abhängigkeiten zu Core-Erweiterungen?
- I7:** Ja, zu der User-Komponente.
- Q:** Ok, das heißt es gibt Abhängigkeiten zu Core-Komponenten. Hast du in diesem Bereich mitentwickelt oder eher an den anderen Views?
- I7:** Mein Part war die Typisierung von statischen und dynamischen Typen.
- Q:** Hast du auch die abhängigen Erweiterungen weiterentwickelt?
- I7:** Funktional nicht. Nur in der Datenbank.
- Q:** Würdest du sagen dass das einfach von der Hand ging oder war das etwas Komplexes?

- I7:** Manuelle Änderungen sind immer aufwendig, wenn man es generieren könnte wäre es schön.
- Q:** Bist du auf Probleme gestoßen welche mit der Architektur von Joomla zu tun haben?
- I7:** Nein eigentlich nicht.
- Q:** War das einfach zu erlernen?
- I7:** Naja, da ich nur weiterentwickeln musste war das kein Problem. Ich musste da nie so tief einsteigen.
- Q:** Hast du schon mal eine eigene View erstellt?
- I7:** Ja
- Q:** Wie bist du da vorgegangen?
- I7:** Copy&Paste und dann halt erweitert.
- Q:** Hast du schon mal irgendeinen Generatoransatz verwendet?
- I7:** Nein bis her nicht.
- Q:** Hast du den Boilerplate-Generator von PHPStorm verwendet?
- I7:** Nein, ich wusste gar nicht das es den gibt.
- Q:** Du kennst ja die Struktur der Komponenten. Wie lange brauchst du für die Entwicklung einer Komponente mit: 6 Views, 3 Entitäten mit Einstiegsview im Backend ohne individuellen Code. Wie lange würdest du dafür brauchen?
- I7:** Naja, schon so ungefähr 3-4 Tage.
- Q:** Wie würdest du dabei vorgehen? Kopieren?
- I7:** Ja, kopieren.
- Q:** Wie testest du die Erweiterung?
- I7:** Wir testen die Erweiterung per Hand.
- Q:** Hat die Komponenten Abhängigkeiten zu 3rd-Party-Erweiterungen?
- I7:** Nein.
- Q:** Wie bist du beim Migrieren von Joomla 2 auf Joomla 3 vorgegangen?
- I7:** Wir haben damals geschaut was sich geändert hat, im Changelog von Joomla. Und sind dann schrittweise vorgegangen die Sachen wieder zum Laufen zu bringen.
- Q:** Habt ihr primär darauf geachtet, dass es funktioniert oder auch Wert darauf gelegt den neuen Joomla 3-Standard einzuhalten?
- I7:** Ich war damals ziemlich neu in der Joomla-Entwicklung und habe nur auf die Funktionalität geachtet. Mein erfahrenerer Kollege hat aber auch auf den Joomla-Standard geachtet.
- Q:** Wie lange habt Ihr den dafür gebraucht?
- I7:** 1 Jahr
- Q:** 40h die Woche?
- I7:** Ja.
- Q:** Was waren den die Hürden am Anfang, als du neu in Joomla warst?
- I7:** Das MVC-Muster wie Joomla es umsetzt und das insgesamt große Joomla-Framework.
- Q:** Hast du auch schon mit andere CMS gearbeitet?
- I7:** Für meine Bachelorarbeit habe ich mal in Drupal und Wordpress reingeschaut. Aber nie großartig drin entwickelt.
- Q:** Neben den Joomla-Konventionen (Namens, Dateistrukturen, Bezeichner), haben dich noch andere Hürden aufgehalten?
- I7:** Das ist im Nachhinein schwer zu sagen. Nein eher nicht.

- Q:** Es gibt 3 große Entwicklungszenarien wenn man mit Joomla arbeitet. Zum einen die Entwicklung von unabhängigen Erweiterungen. Die Entwicklung von Erweiterungen mit Abhängigkeiten. Der dritte Anwendungsfall ist die Migration. Würdest du sagen das sind alle oder fallen dir noch mehr ein?
- I7:** Nein ich denke das waren alle.
- Q:** Bei welchen Anwendungsfällen, denkst du ist ein Generatoransatz sinnvoll?
- I7:** Auf jeden Fall beim Ersten. Also Neuentwicklung, dass ist immer von Vorteil wenn man das Grundgerüst generieren lassen kann. Dann hat man eine einheitliche Codequalität, alles funktioniert und installierbar. Darauf kann man dann aufbauen. Was war nochmal der zweite Anwendungsfall?
- Q:** Der zweite war die Entwicklung von abhängigen Erweiterungen.
- I7:** Generell finde ich es von Vorteil wenn man sich etwas generieren lassen kann.
- Q:** Wie sieht es im Anwendungsfall, der Migration aus?
- I7:** So habe ich mir das für Groups vorgestellt. Weil ja jetzt auch in Joomla 4 Namespaces verwendet werden.
- Q:** Welche Features würdest du von einem modelgetriebenen Ansatz erwarten?
- I7:** Sehr einfach neue Views der vorhandenen Komponente hinzufügen. Mehr fällt mir jetzt nicht ein.
- Q:** Würdest du einen textuellen oder visuellen Modellierungsansatz bevorzugen?
- I7:** Über in Modellierungswerkzeug wie UML.
- Q:** Also visuell und nicht textuell. Würdest du dir einen Dialog oder Wizard wünschen?
- I7:** Muss nicht unbedingt sein.
- Q:** Ok, vielen Dank.

Interviewee 8

Interviewer: *Wolf Rost*, Date: *03.05.2018*, Location: *Gießen, Germany*

- Q:** Als erstes will ich dich drauf Hinweis, dass wenn wird das hier veröffentlichen, wir die Daten natürlich anonymisieren. In so fern du das willst.
- I8:** Mir egal.
- Q:** Die erste Frage an dich. Hast du schon einmal Joomla-Erweiterungen entwickelt und wenn ja für welche Versionen?
- I8:** Ja, für 1.5, 1.6 und so weiter. Seit 1.5.
- Q:** Also kann man sagen seit 1.5 bis 3.8.
- I8:** Genau.
- Q:** Wie viele Erweiterungen hast du entwickelt?
- I8:** Um die 40. Erweiterungen die es noch gibt 15.
- Q:** Das heißt von den 40 sind jetzt noch 15 im Einsatz.
- I8:** Ja es sind einige dadurch das Joomla sich weiterentwickelt hat, sind sie unnötig geworden. Ich habe auch auf einige keine Aufsicht mehr.
- Q:** Noch einmal eine Frage zu den 40 Erweiterungen. Was waren das den für Typen von Erweiterungen? Kannst du quantifizieren wieviele davon Komponenten, Module...?
- I8:** 2 Hauptkomponenten, Organizer und Groups. Groups kam aber auch so dazu, da war ich nicht von Anfang an dabei. Bibliotheken, ich habe das eine entwickelt und zwei davon in den Sand gesetzt. Templates, zwei Templates selbst entwickelt, eins weiterentwickelt. Plugins, etliche. Diese Knöpfe, die Content-Plugins, die die Sachen von den Knöpfen auflösen
- Q:** Also die ganzen Editor-Plugins. Vielleicht anderst, wie viele Module waren das? Kannst du das eher sagen, weil das einfacher ist? Oder weniger?
- I8:** Auch da, so um die 15.

- Q:** Dann nochmal speziell auf die Komponenten eingegangen. Wie viele Views waren das ungefähr? Kannst du das noch unterteilen in Frontend und Backend? Von deinen zwei Komponenten.
- 18:** Ich denke mal um die 50.
- Q:** Insgesamt oder nur Frontend/Backend?
- 18:** Achso dann so 60-65. Aber im Backend, gerade weil ich es sehr allgemein entwickelt habe. Obwohl es schon reichlich viele Views sind, ist es eher im Endeffekt so wie 3 oder 4. Weil ich habe meine Edit-View, meine Lists und meine Merge-View. Und ich verwende mein Werkzeug um neue Views auszustellen. Klar ich habe ganz viele aber im Endeffekt ist der Quellcode sehr schmal. Weil sich da so vieles wiederholt im Backend.
- Q:** Weil du das dann ausgelagert hast?
- 18:** Ja
- Q:** Ah ok. D.h. du konfigurierst nur noch, rufst etwas auf und es erstellt dir eine View?
- 18:** Ja, im Endeffekt ich habe ein Model ohne Code, ich erbe alles vom Elternteil und ich geb da nur ein paar Parameter zu und es läuft.
- Q:** Also 65 Views sind es Insgesamt obwohl es im Backend nur drei sind aber...
- 18:** Etliche ausführung.
- Q:** Das klingt ja sehr interessant. Dadurch dass du das auslagert, wie viel Prozentsatz denkst du den sind dann so standard CRUD views.. Also so listen-views mit CRUD-Funktionalitäten und Detail-View, von diesen 65.
- 18:** Das ist halt sinn und zweck von backend.
- Q:** Also im Backend würdest du sagen, 100
- 18:** Also wenn die Frage eher so gemeint ist wieviel ich gespart habe, das kannst du eigentlich direkt nachvollziehen, wenn du irgendwo reinschaust.
- Q:** Also mir ging es erstmal nur um die Art der Views und das wären dann also...
- 18:** List, Edit und Upload/Merge aber das Upload ist eher so ein Edit.
- Q:** Also sind es eigentlich 100
- 18:** Ehm, auch Batch, das habe ich vergessen. Das ist in Groups. Hätte ich eigentlich gerne im Organizer aber ich hatte keine Zeit das anzugehen.
- Q:** Ok, und Batch ist dann individuell entwickelt?
- 18:** Ehm, je nach Batch. Das habe ich angepasst, diese Sache von Groups, die bilden assoziationen zwischen x-Dings, so dass man erstmal eine Sache auswählt und dann mit dieser Sache die man ausgewählt hat etwas hinzufügt und dann noch etwas. Das ist sehr ausgefallen, aber in der Regel ist es auch eine Listen-View.
- Q:** Ok, dann passt das mit den 100
- 18:** In der Regel nicht, ich halte so etwas eher für Unsinn.
- Q:** D.h. entweder hast du gar keine Crossreferenzen oder die Crossreferenz geschieht dann über Buttons?
- 18:** Ich weiss nicht, eher so ein Feldtyp oder so. Bei Curriculum, wir haben Studiengänge und man kann dann Module und Modulpools hinzufügen, wenn man auf die klickt kann man da hin gehen um sie zu editieren. Allerdings ist dies nicht unbedingt...Ich denke die wenigsten machen das. Weil ansich man möchte nur eine Hierarchy aufbauen und da braucht man eigentlich nur die Referenz auf sachen und nicht die Sachen zu editieren ansich....das könnte ich eigentlich ausbauen...
- Q:** Du hast ja jetzt schon gesagt das 100
- 18:** Im Backend
- Q:** Im Backend, genau. D.h. dein Anteil von individuellem Code im Backend ist dann wahrscheinlich sehr gering, wenns darum geht die Darstellung die Views aufzubauen.
- 18:** Ja es gibt nur ganz spezifische Views, wie der Stundenplan-Upload. Da hängt einiges dran.Es gibt sachen die der Regel entsprechen und das ist dann 100
- Q:** Das ist dann aber individuell im Sinne der Businesslogik oder?

- 18:** Businesslogik, wie die Sachen interiert sind, welche zusammenhänge sie haben oder ich schaue auf die Modulnummer und dann such ich mit dieser Modulnummer in dem Kontext von Studiengang ob ich da eine Zuordnung machen kann. und das geschieht alles im Hintergrund.
- Q:** Wie ist den dein Vorgehen wenn du eine neue Komponente entwickelt müsstest? Schaust du dir irgendwo beispiele an? Fängst du von Null an und erstellst die Dateien komplett neu? Oder hast du einen Generator? machst du Copy&Paste von altem Zeug. Wie ist den da dein generelles Vorgehen?
- 18:** Mein generelles Vorgehen wäre erst einmal zu Überlegen welche Entitäten ich habe. Weil dies ist sehr entscheidend wie die Sachen die Verwalt werden zu einander stehen. Dann muss ich überlegen wie die Kardinalitäten sind. Ich finde dieses Zusammenspiel spielt eine entscheidende Rolle wie alles im Endeffekt aussieht. Erstmal das festlegen und dann Datenbank. Dann erstelle ich einzelne Funktionen im Model, welche die Sachen machen, mit den Daten, die ich für wichtig halte. Also diese CRUD-Geschichte von wegen anlegen, löschen, editieren, muss da sein. Aber auch wie verbindet man wie sieht das aus? Ich finde gerade diese Verbindungsgeschichte die fehlt in Joomla. Die machen sehr große Fortschritte bei der Datenbank, also diese Datenbankabstraktion haben sie sehr gut gemacht. Aber mit MyISAM, da haben sie nicht so an Foreign-Keys gedacht. Und das fehlt.
- Q:** Vielleicht nochmal anderst gefragt. Wie würdest du die Komponente an sich selbst umsetzen. Wie würdest du da Anfangen die Views zu entwickeln? Das ganze Grundgerüst einer Komponente aufzubauen?
- 18:** Da würde ich tatsächlich auf Copy&Paste zurückgreifen. Zu mal es von Joomla gefordert wird. Man muss ein Manifest haben, man muss bestimmte Dateien haben, diese müssen eine bestimmte Struktur haben.
- Q:** Wie siehst bei Modulen aus?
- 18:** Das komplette Ding, Copy&Paste, die Namen editieren und dann die richtigen Sachen angehen.
- Q:** Du hast Komponenten und Module entwickelt. Wahrscheinlich haben dann deine Module auch Abhängigkeiten zu den Komponenten?
- 18:** Ja
- Q:** Weil sie die Daten anzeigen oder nutzen? Was machen die Module mit den Komponenten.
- 18:** In der Regel gehen sie auf die Daten um etwas darzustellen oder sprechen AJAX-Funktionen an, wenn es dynamisch gehen soll. Aber diese AJAX-Funktionen habe ich dann als Views in der Komponente, also wirklich eine AJAX-View die dann angesprochen wird.
- Q:** Ok, gehen deine Module direkt auf die Datenbank-Tabellen oder benutzen sie, das Module oder eine View der Komponente?
- 18:** Kommt ganz drauf an, ich versuche so viel wie Möglich, die Abfragen in der Komponente zu machen, so dass man sicherstellen kann diese noch funktionieren damit ich sicherstellen kann das wenn ich etwas in der Komponente ändere das Modul noch funktioniert. Manchmal lässt sich das nicht vermeiden aber dadurch das vieles in der Komponente geschieht...man meidet eingestiegen in Plugins und Modulen.
- Q:** So generell wenn du eine Komponente entwickelt müsstest, die installierbar und eine Frontend- und Backend-View hat. Wie lange brauchst du dafür? Mit deinem Clone-and-Own approach...also die muss nur installierbar sein keine große Businesslogik.
- 18:** 10 Minuten
- Q:** 10 Minuten?
- 18:** Ja, im Endeffekt muss ich nur kopieren und einiges Löschen aber wenn ich entschlagte habe kann ich dann per RegExpression oder String-Suche, einfach ersetzen. Das ist eine Sache von paar Minuten.
- Q:** Wenn du schon eine Komponente hast und diese nur um eine View erweitern musst. Wie gehst du da vor?
- 18:** Da kopiere ich eine ähnliche vorhandene View, also wenn es Backend ist, und pass es an. Ich habe sehr viel ausgelagert...ich brauche dazu 3 Minuten.
- Q:** Wie sieht es bei einem neuen Modul aus?
- 18:** Kommt drauf an was es tun soll
- Q:** Ein Modul was Daten von einer Komponente anzeigen soll.
- 18:** Also einfach eine Liste?
- Q:** Genau, z.B. der Entitäten. Also erstmal, wie gehst du wieder vor? Hast du einen Generator oder Boilerplate-Code?

- I8:** Hast du eigentlich schon gefragt bei einer anderen Frage... Also bei Modulen würde ich einfach ein vorhandenes Modul nehmen und kopieren.
- Q:** Also Copy&Paste und anpassen?
- I8:** Ja
- Q:** Wie sieht es bei Template-Overrides aus? Also wenn du eine View neu gestalten willst?
- I8:** Die meide ich, so gut es geht.
- Q:** Also benutzt du sie garnicht?
- I8:** Ich habe die Erfahrung, dass sich dadurch Fehler einschleichen, wenn die Komponente oder...welche Extension auch immer sich anpasst. Ich meine gerade da, da geschieht soviel im Model oder View und man kann eigentlich nur das Template...mit bestimmten Ausnahmen. Alleine wenn sich die Variablennamen ändern...alles verloren. Ich verstehen warum man Template-Overrides benutzt, ich würde es aber nicht machen.
- Q:** Ok. Hast du schon einmal versucht, das Model (DAO) einer 3rd-Party-Erweiterungen in deiner eigenen Komponente verwendet? Um z.b. Daten anzufragen.
- I8:** Das hatte ich vor langer Zeit gemacht als Groups noch nicht meine Sache war.
- Q:** Und wie hast du die Daten da angefragt? Bist du direkt auf die Datenbank gegangen oder hast du versucht das Model von Groups zu verwenden?
- I8:** Ich bin in meiner Komponente auf die Tabelle von der Anderen um mir einen Link zusammenzubauen. Das war statisch ohne Ende und das hat aber auch nicht lange überlebt.
- Q:** Also den Fall dass du das Model benutzt hast gab es nicht.
- I8:** Ich habe das bei den damaligen Groups-Entwicklern angefragt, aber in 3 Jahren hat es nicht geklappt.
- Q:** D.h. du hättest es gerne benutzt anstatt auf die Datenbank zugehen?
- I8:** Nein, ich wollte eigentlich nichts mit Groups nichts zu tun haben.
- Q:** Hattest du schon mit der Migration von Komponenten zu tun?
- I8:** Ja
- Q:** Kannst du auch sagen von welchen Versionen?
- I8:** Von 1.5 auf 1.6/1.7 und dann auf 2,3.
- Q:** Wieviele Erweiterungen musstest du da so migrieren?
- I8:** Nur meine Sachen, höchstens 15...eher 10.
- Q:** Wie bist du beim migrieren vorgegangen? Konntest du dich irgendwo orientieren? Hast du die Dokumentation gelesen, gab es ein Tool dass das für dich gemacht hat?
- I8:** Ich habe die Dokumentation gelesen und ich hab einfach alles angepasst, wo die anweisung von Joomla waren, das musst du anpassen. Und dann habe ich ausprobiert bis es ging. Also die Fehler nach und nach behoben.
- Q:** Also es gab einen Migration-Guide von Joomla dem du folgen konntest?
- I8:** Ist schon sehr lange her, ich sage mal ja. Es gab auf jedenfall hinweise.
- Q:** Kannst du dich noch dran erinnern wie lange du für die 10 Erweiterungen gebraucht hast?
- I8:** Ich weiß nicht mehr genau, das war auch um die Zeit wo wir die Entwicklerinfrastruktur angeschafft haben. Also mit Metriken, Copy&Paste, Zyklische Komplexität, etc. Ich würde sagen 6 Monate so grob. Allerdings kann es sein das ich da auch ein wenig dass mit den Metriken reinmische.
- Q:** Also deine Codequalität verbessert hast?
- I8:** Ja ich weiß nicht mehr ob das ein und der selbe Vorgang war oder getrennt.
- Q:** Also ca. 6 Monate. Hast du auch schon versucht Erweiterungen von einer anderen Plattform oder zu einer anderen Plattform, wie Wordpress zu migrieren?
- I8:** Nein, noch garnicht.
- Q:** Es gibt in Joomla ja Konventionen. Man muss sich an die Dateistruktur halten und bestimmte Namen/Bezeichner verwenden. Was denkst du, wie häufig das die Fehlerquelle war.

- 18:** Das hat mir nicht wirklich Steine in den Weg gelegt. Wo ich große Probleme hatte, war mit Sprachkonstanten in dem Manifest. Es gibt einen Dateiname, den Komponentennamen und einen anderen Namen im XML und eine Beschreibung. Ich weiß besonders bei den Plugins, da musste die eine Datei der anderen Datei gleichen und dann auch in der Sprachdatei stehen oder auch nicht...es war sehr verwirrend.
- Q:** Das war auf jedenfall ein Punkt wo dir die Joomla-Konventionen, Steine in den Weg gelegt hat?
- 18:** Ja
- Q:** Vielleicht noch andere Sachen die Joomla eigen macht oder worüber man häufig stolpert?
- 18:** Diese Konventionen, ich bin einer der versucht immer alles Richtig zu machen. Auch wenn das ein bisschen Ansichtssache ist. Einige der Konventionen von wegen Codestyle. Ich nehme das von PHPStorm und die sprechen auch mit einander. Aber dann stimmte das von PHPStorm nicht hundertprozentig mit dem von Joomla. Oder es ist von Joomla nachträglich etwas gekommen oder PHPStorm hat es nicht angepasst. Und dann halt diese PSR2/4-Geschichte. Ich habe alles angepasst aber dann sollte es doch nicht so sein...ich mochte den Joomla-Style mehr als den von PSR2, mit Ausnahme der Tabs. Es wäre nett wenn sie ein für alle mal entscheiden und es dann belassen. Ich kann mit PSR2 leben auch wenn es nicht so schön ist. Aber dass das immer wieder diskutiert werden muss...Wie macht ihr es in Joomla?
- Q:** Wir halten uns PSR4 wegen den Namespaces und allem was in Joomla 4 reinkommen soll. Kennst du Generatoransätze die dir ja Arbeit abnehmen sollen? Im Bereich von Joomla?
- 18:** Ich habe von eurem Werk gehört allerdings noch nicht verwendet.
- Q:** Du weißt aber ungefähr was es kann? Wo für man es einsetzen kann?
- 18:** Ja für neues Projekt oder halt vorhanden migrieren.
- Q:** Denkst du auch das sind sinnvolle Szenarien? Bzw. wo so ein Generatoransatz helfen könnte?
- 18:** Ja, bestimmt.
- Q:** Fällt dir ein Szenario ein wo ein Generatoransatz garnicht passen würde?
- 18:** Beim Hochladen des Stundenplan oder importieren der Sachen aus LSF.
- Q:** Das sind also individuelle Sachen?
- 18:** Ja
- Q:** Und da kannst du dir einen Generatoransatz schwer vorstellen.
- 18:** Ja, nur für das Gerüst aber nicht für den Code an sich.
- Q:** Also die Businesslogik, also für ein Scaffolding schon. Was für Funktionalität oder Features erwartest du von so einem Generatoransatz?
- 18:** Neue Extension erstellen: Handling der Entitäten.
- Q:** Also das du die Entitäten beschreiben kannst und die diese dann auch generiert werden? Die Tabellen z.B.
- 18:** Ja. Inklusive Foreign-Keys und Constrains. Das ist das A und O. Diese CRUD-Ansichten müssten eigentlich auch locker von der Hand gehen. Die Frontend-Ansichten da sehe ich eher Schwierigkeiten, es sei den es ist auch einfach nur eine Darstellung von Listen oder individuellen Attributen. Kann man Templates über Joomla machen?
- Q:** Nein, aber das wäre etwas was du erwartest?
- 18:** Wenn dann den schon, aber da sehe ich auch sehr große Schwierigkeiten weil es sehr grafisch ist. Und man muss da vieles Bedenken, das nichts mit den anderen Extensions zu tun hat.
- Q:** Also wenn ich das nochmal zusammenfassen kann. Du denkst so Standard-Views, dass sollte das Tool auf jedenfall können. Listen/Detail-Ansichten die ganzen CRUD-Funktionalitäten.
- 18:** Es gibt viele Module, die Listen oder einzelne Dinge anzeigen. Das wäre genau so eine View wie aus der Komponenten nur halt dynamisch einzubinden und das müsste eigentlich ein Klacks sein für euch.
- Q:** Also Liste-/Entitäten-Ansicht
- 18:** Ja aber das würde ich quasi von euch für die Module als Template erwarten. Das man sagt was habe ich da für ein Modul. Habe ich ein Listen-Modul, habe ich ein Item-Modul, habe ich ein Such-Modul, einfach ein paar Template und wie das Modul mit den Daten hantieren würde. Die Buttons bzw. die Plugins, also ähnliche Geschichte.
- Q:** Ehm, bei Buttons, du redest dann von Editor-Buttons?

I8: Genau von editor-xttd. Und auch System- und Content-Plugins. Auch da würde ich unterschiedliche Template erwarten für Plugins. Und da auch unterschiedliche Templates würde ich erwarten für Plugins.

Q: Also unterschiedliche Templates.

I8: Von wegen was soll diese Plugin tun und wie soll es aussehen.

Q: Also dass man auch das Aussehen schon mit beschreiben kann. Ok, vielen Dank.

B

Meta-Model of the cJSL DSL

Here, the meta-model parts of the cJSL DSL is presented in detail.

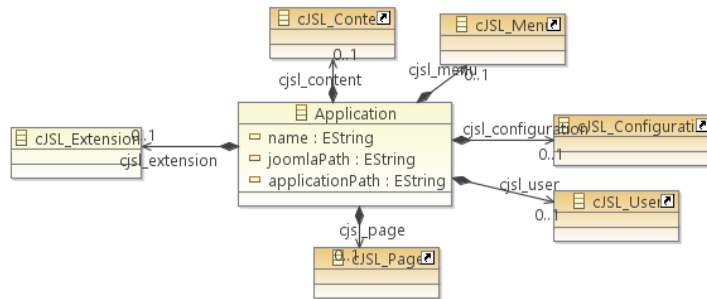


Figure B.1: cJSL Application with all cJSL Parts

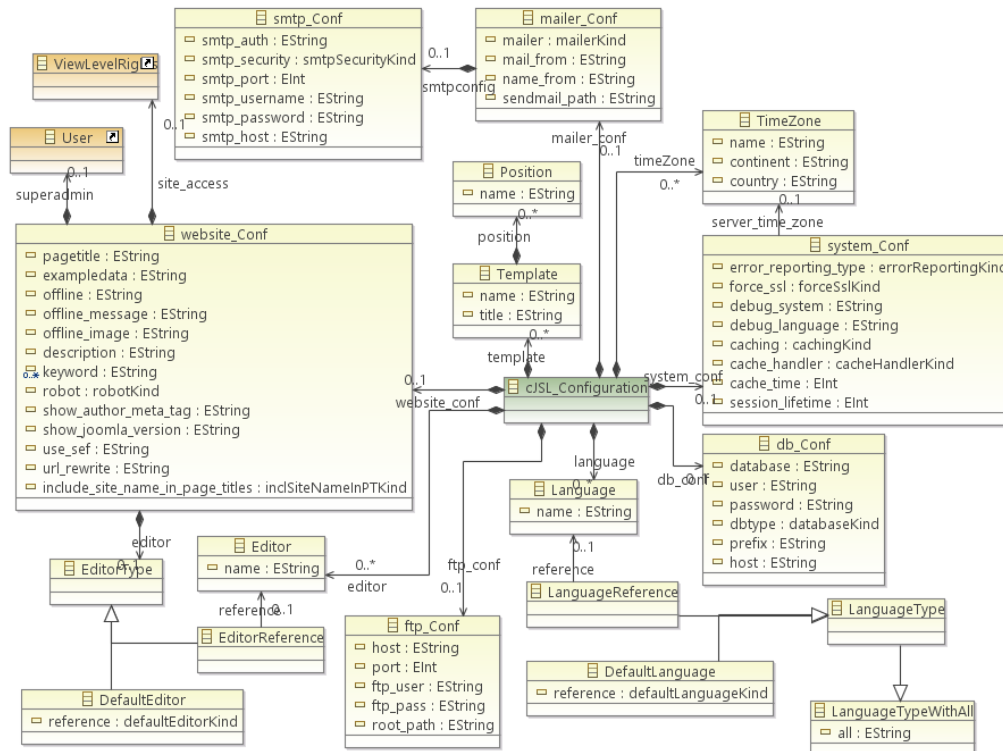


Figure B.2: cJSL Configuration Part

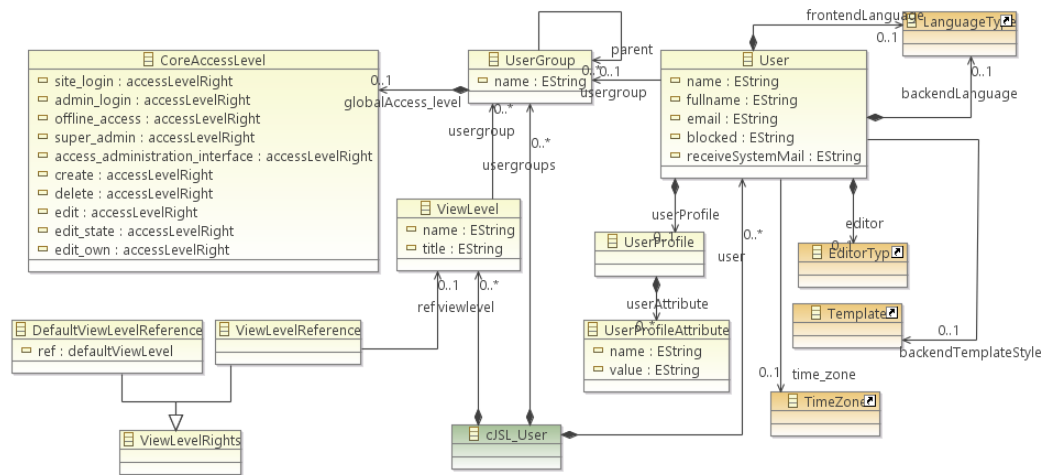


Figure B.3: cJSL User Part

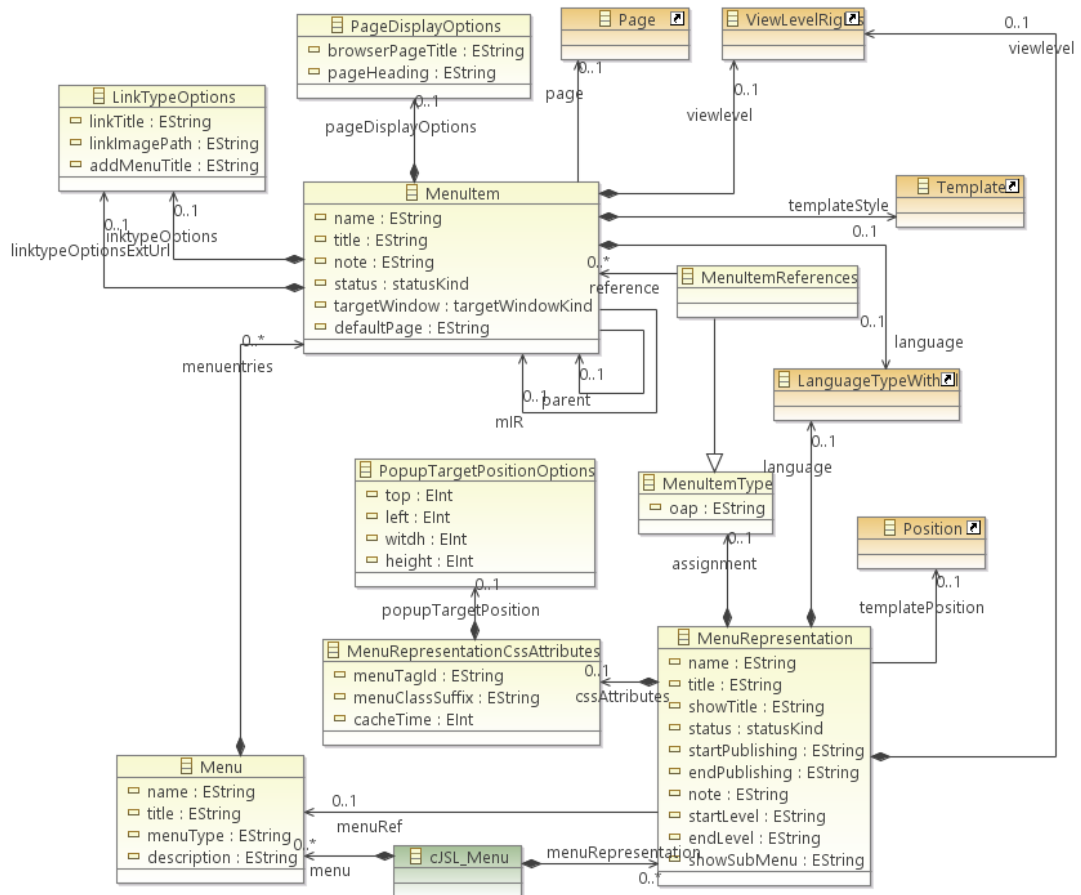


Figure B.4: cJSL Menu Part



Figure B.5: cJSL Content Part

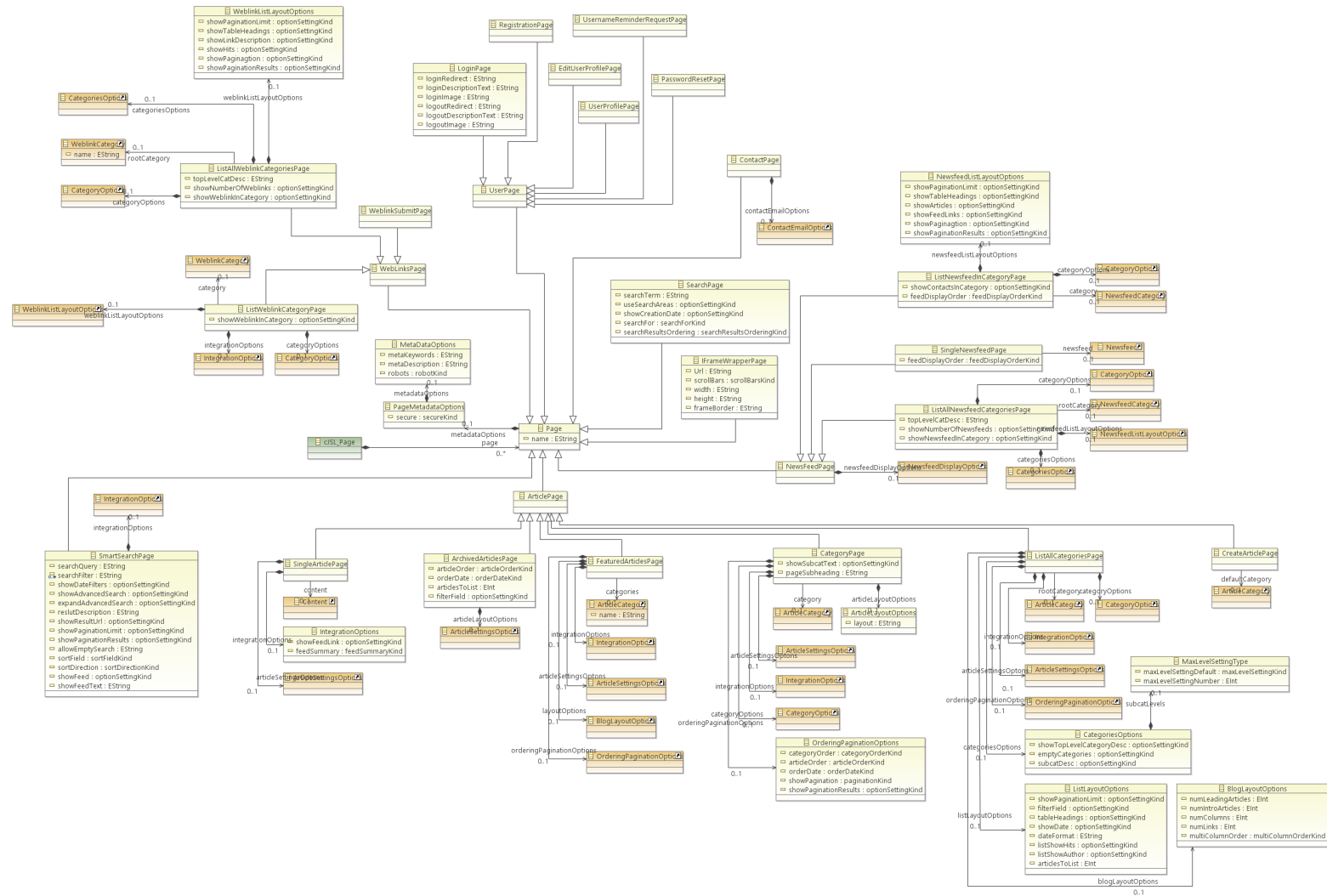
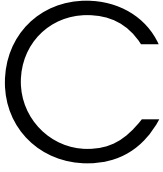


Figure B.6: cJSL Page Part



Well-Formedness Rules for eJSL

Here, all constraints for the eJSL language are collected.

Entities

```
1 context Entity
2
3 — Check if entity names are unique
4 inv uniqueEntityIdentifier:
5 Entity.allInstances()->collect('^.concat(name))->union(Entity.allInstances()->
6 collect(name))->isUnique(i|i)
7 — Alternatively: Entity.allInstances()->forAll(e1, e2| e1 <> e2 implies e1.
8 name <> e2.name and e1.name <> '^' + e2.name)Entity.allInstances()->forAll(
9 e1, e2| e1 <> e2 implies e1.name <> e2.name and e1.name <> '^' + e2.name)
10
11 — Check if attribute identifiers are unique
12 inv uniqueAttributeIdentifier:
13 self.attributes->collect('^.concat(name))->union(self.attributes->collect(name
14 ))->isUnique(a|a)
15 — Alternatively: self.attributes -> forAll(a1, a2| a1 <> a2 implies a1.name <>
16 a2.name and a1.name <> '^' + a2.name)
17
18 /* The noGeneralizationCycles constraint includes this one
19 — Check if entity does not extend itself
20 inv entityDoesNotExtendItself:
21 self.supertype -> excludes(self)
22 */
23
24 — Check if at least one unique (primary) attribute exists if entity does not
25 extend another one. If so, check if closure of parent entities has a unique
26 attribute
27 inv primaryAttributeExists:
28 self.attributes -> exists(isunique) or self.supertype->closure(supertype).
29 attributes-> exists(isunique)
30
31 — Check if no generalization cycle exists (transitive closure)
32 inv noGeneralizationCycles:
33 self.supertype->closure(supertype)->excludes(self)
```

Listing C.1: Well-Formedness-Rules: Context Entities

```
1 context StandardTypes
2 — Check if AutoIncrement is used for suitable types
3 inv autoIncrementForInteger:
4 self.type=StandardTypeKinds::Integer implies self.autoincrement
```

Listing C.2: Well-Formedness-Rules: Context StandardTypes

```

1  context Reference
2  — Check if min/max values are -1, 0, and 1
3  inv allowedMinValues:
4  Set{'0','1'} -> includes(self.lower)
5
6  — Check if min/max values are -1, 0, and 1
7  inv allowedMaxValues:
8  Set{'1','-1'} -> includes(self.upper)
9
10 — Check if referenced entity exists
11 inv referencedEntityExists:
12 Entity.allInstances().name->includes(self.entity.name)
13
14 — Check consistency of referencedEntity and referencedEntityAttribute
15 inv consistentEntityAttribute:
16 self.entity.attributes.name
17 ->includes(attributereferenced.name->first())
18
19 — Check if references have no cycles
20 inv noReferenceCycles:
21 self.entity.references
22 -> closure(entity.references)
23 -> forAll(r|r.attributereferenced <> self.attribute)
24
25 — Check if attribute types are consistent in references
26 inv consistentAttributeTypes:
27 self.attribute.type.oclAsType(StandardTypes).type.oclAsType(StandardTypeKinds)
28 =
29 self.attributereferenced.type.oclAsType(StandardTypes).type.oclAsType(
30   StandardTypeKinds)
31
32 — Check if reference is a primary (unique) attribute
33 inv referenceIsPrimary:
34 self.attributereferenced -> exists(isunique) or self.attributereferenced ->
35   exists(isprimary)

```

Listing C.3: Well-Formedness-Rules: Context Reference

Pages

```

1  context Feature
2  — Check if page uses an entity only once
3  inv useEntityOnlyOnce:
4  self.entities->forAll(e| self.entities->count(e) = 1)

```

Listing C.4: Well-Formedness-Rules: Context Feature

```

1  context DetailsPage
2  — Check if Text mapped to Textarea
3  inv detailsPageFieldTextarea:
4  self.editfields.attribute.type.oclAsType(StandardTypes).type.oclAsType(
5    StandardTypeKinds) =
6    StandardTypeKinds::Text->asSequence()
7  implies
8  self.editfields.htmltype.oclAsType(HTMLTypes).oclAsType(SimpleHTMLTypes).
9    htmltype.oclAsType(SimpleHTMLTypeKinds) =
10   SimpleHTMLTypeKinds::Textarea->asSequence()

```

Listing C.5: Well-Formedness-Rules: Context DetailsPage (1)

```

1 context DetailsPage
2 — Check if Short_Text and Link mapped to Text_Field
3 inv detailsPageFieldText_Field :
4 let attrType : Sequence =
5   self.editfields.attribute.type.oclAsType(StandardTypes).type.oclAsType(
6     StandardTypeKinds)
7   in attrType = StandardTypeKinds::Short_Text->asSequence()
8   or attrType = StandardTypeKinds::Link->asSequence()
9 implies
10  self.editfields.htmltype.oclAsType(HTMLTypes).oclAsType(SimpleHTMLTypes).
11    htmltype.oclAsType(SimpleHTMLTypeKinds) =
12    SimpleHTMLTypeKinds::Text_Field->asSequence()
13 — Check if Boolean mapped to Yes-No-Buttons
14 inv detailsPageFieldBoolean :
15 self.editfields.attribute.type.oclAsType(StandardTypes).type.oclAsType(
16   StandardTypeKinds) =
17   StandardTypeKinds::Boolean->asSequence()
18 implies
19  self.editfields.htmltype.oclAsType(HTMLTypes).oclAsType(SimpleHTMLTypes).
20    htmltype.oclAsType(SimpleHTMLTypeKinds) =
21    SimpleHTMLTypeKinds::Yes_No_Buttons->asSequence()
22 — Check if Integer mapped to Input Integer
23 inv detailsPageFieldInteger :
24 self.editfields.attribute.type.oclAsType(StandardTypes).type.oclAsType(
25   StandardTypeKinds) =
26   StandardTypeKinds::Integer->asSequence()
27 implies
28  self.editfields.htmltype.oclAsType(HTMLTypes).oclAsType(SimpleHTMLTypes).
29    htmltype.oclAsType(SimpleHTMLTypeKinds) =
30    SimpleHTMLTypeKinds::Integer->asSequence()
31 — Check if Time, Date and Datetime mapped to Datepicker
32 inv detailsPageFieldDatepicker :
33 let attrType : Sequence =
34 self.editfields.attribute.type.oclAsType(StandardTypes).type.oclAsType(
35   StandardTypeKinds)
36   in attrType = StandardTypeKinds::Time->asSequence()
37   or attrType = StandardTypeKinds::Date->asSequence()
38   or attrType = StandardTypeKinds::Datetime->asSequence()
39 implies
40  self.editfields.htmltype.oclAsType(HTMLTypes).oclAsType(SimpleHTMLTypes).
41    htmltype.oclAsType(SimpleHTMLTypeKinds) =
42    SimpleHTMLTypeKinds::Datepicker->asSequence()
43 — Check if Image mapped to Imagepicker
44 inv detailsPageFieldImagepicker :
45 self.editfields.attribute.type.oclAsType(StandardTypes).type.oclAsType(
46   StandardTypeKinds) =
47   StandardTypeKinds::Image->asSequence()
48 implies
49  self.editfields.htmltype.oclAsType(HTMLTypes).oclAsType(SimpleHTMLTypes).
50    htmltype.oclAsType(SimpleHTMLTypeKinds) =
51    SimpleHTMLTypeKinds::Imagepicker->asSequence()
52 — Check if File mapped to Filepicker
53 inv detailsPageFieldFilepicker :
54 self.editfields.attribute.type.oclAsType(StandardTypes).type.oclAsType(
55   StandardTypeKinds) =
56   StandardTypeKinds::File->asSequence()
57 implies
58  self.editfields.htmltype.oclAsType(HTMLTypes).oclAsType(SimpleHTMLTypes).
59    htmltype.oclAsType(SimpleHTMLTypeKinds) =
60    SimpleHTMLTypeKinds::Filepicker->asSequence()

```

Listing C.6: Well-Formedness-Rules: Context DetailsPage (2)

```

1  context IndexPage
2  — Check if linked attribute is consistent to entity in IndexPage
3  inv linkedIndexAttributeConsistent:
4  self.links->forAll(l | self.entities.attributes.name->includes(l.
    linkedAttribute.name) and self.tablecolumns.name->includes(l.
    linkedAttribute.name))
5
6  — Check if table columns and filters are used only once in a page
7  inv tableColumnsFiltersOncePerPage:
8  self.filters->forAll(f | self.filters->count(f) = 1) and self.tablecolumns->
    forAll(tc | self.tablecolumns->count(tc) = 1)
9
10 — Check if filters are a subset of representation columns in IndexPage
11 inv filtersIncludedInTableColumns:
12 self.filters->forAll(f | self.tablecolumns->includes(f))
13
14 — Check if entities and attributes are defined, if filters are using them
15 inv FilterAttributesAndEntitiesDefined:
16 self.filters->forAll(f | Entity.allInstances()->includes(f.oclContainer()) and f
    .oclContainer().oclAsType(Entity).attributes->includes(f))
17
18 — Representation columns and filters must be consistent to *Entity in IdexPage
19 inv RepColsandFiltersConsistent:
20 self.tablecolumns->forAll(tc | self.entities->includes(tc.oclContainer()) and
    tc.oclContainer().oclAsType(Entity).attributes->includes(tc))
21 and self.filters->forAll(f | self.entities->includes(f.oclContainer()) and f.
    oclContainer().oclAsType(Entity).attributes->includes(f))
22
23 — If a page uses more than one Entity, there can only be one main Entity that
    is not referenced by all other Entities
24 inv MultiplePageEntityReferences:
25 let mainEntity : Sequence(Entity) = self.entities->select(e | self.entities.
    references.entity->excludes(e))
26 in entities->size() > 1
27 implies
28 mainEntity->size() = 1

```

Listing C.7: Well-Formedness-Rules: Context IndexPage

Extensions

```

1  context Library
2  — Check if library classes are unique
3  inv uniqueLibClass:
4  self.classes->notEmpty() implies self.classes->forAll(c1, c2 | c1 <> c2
    implies c1.name <> c2.name)

```

Listing C.8: Well-Formedness-Rules: Context Library

```

1  context Class
2  — Check if class method names are unique in libraries
3  inv uniqueLibMethNames:
4  self.oclContainer().oclIsTypeOf(Library) implies self.methods->forAll(m1, m2 | m1
    <> m2 implies m1.name <> m2.name)

```

Listing C.9: Well-Formedness-Rules: Context Class

```

1  context CMSExtension
2  — Check if language keys are unique for one language definition
3  inv uniqueLangKeys:
4  self.extensions.languages->forAll(l | l.keyvaluepairs->forAll(kvn1, kvn2 | kvn1
   <> kvn2 implies kvn1.name <> kvn2.name))
5
6  — Unique extension names for extensions of same type
7  inv uniqueExtNames:
8  self.extensions->forAll(e1, e2 | e1 <> e2 and e1.ocIsTypeOf(e2.ocType())
   implies e1.name <> e2.name)
9
10 — Check for valid data mapping for modules and components (is component
   existing...)
11 inv validCompModMapping:
12 self.extensions->
13 forAll(e | e.ocIsTypeOf(Module) and e.ocAsType(Module).pageRef->notEmpty())
14 implies Component.allInstances().name->includes(e.ocAsType(Module).pageRef.
   pagescr.ref.name)
15 )
16
17 — Check if not more than one backend and frontend is defined
18 inv atomicBackFrontEnd:
19 self.extensions->forAll(e | e.ocIsTypeOf(Component)
20 implies e.ocAsType(Component).sections->select(s | s.ocIsTypeOf(
   FrontendSection))->size() <= 1 and
21 e.ocAsType(Component).sections->select(s | s.ocIsTypeOf(BackendSection))->
   size() = 1
22 )

```

Listing C.10: Well-Formedness-Rules: Context CMSExtension

D

Generator Scalability Tests

In the following, we illustrate the settings and present the original results of the generator scalability tests.

Scalability Test 1: Scaling the Number of Generated Components

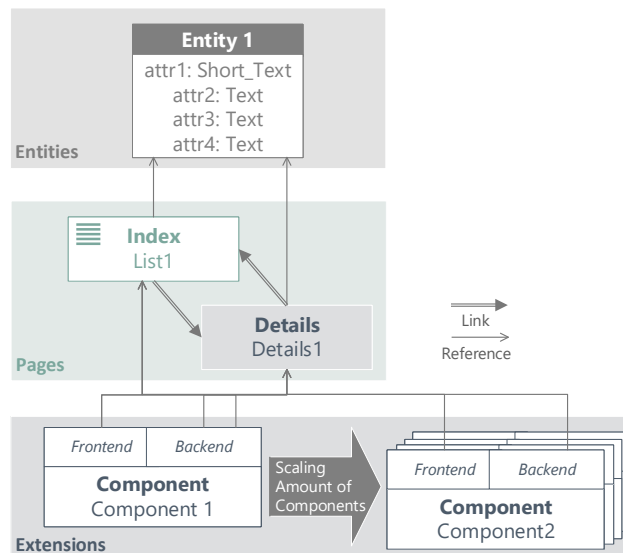


Figure D.1: Measurement Setting (Test 1)

Table D.1: Measurement Setting (Test 1)

Model #	Components	Index Pages	Details Pages	Referenced Pages per Component	Entities without References
1	1	1	1	4	1
2	3	1	1	4	1
3	5	1	1	4	1
4	8	1	1	4	1
5	20	1	1	4	1
6	40	1	1	4	1
7	70	1	1	4	1
8	100	1	1	4	1

Table D.2: Measurement Result (Test 1)

Model #	CPU Time (s)	RAM (MB)	Generated Files	Package Size (MB)
1	0.39	0.9	60	0.36
2	0.83	2.6	180	1.10
3	1.34	4.5	300	1.80
4	1.92	7.1	480	2.90
5	5.16	17.6	1200	7.20
6	9.89	35.4	2400	15.00
7	16	62.3	4200	25.00
8	25.90	92	6000	36.00

Scalability Test 2: Scaling the Number of Pages

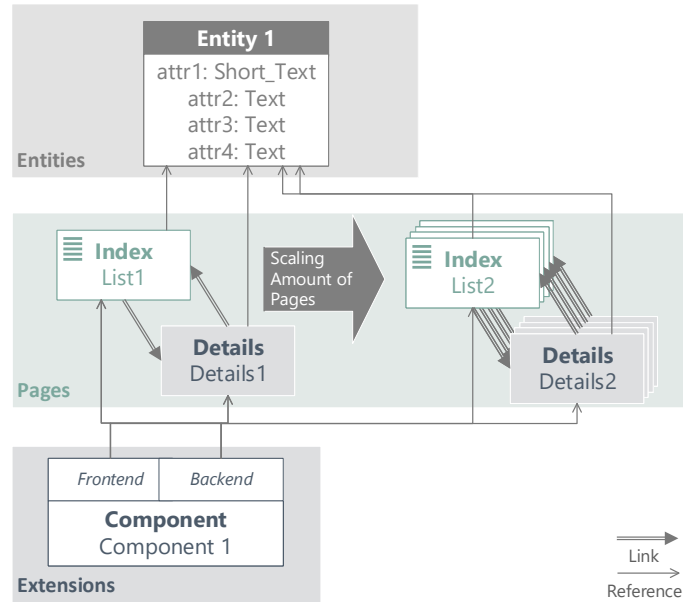


Figure D.2: Measurement Setting (Test 2)

Table D.3: Measurement Setting (Test 2)

Model #	Components	Index Pages	Details Pages	Referenced Pages per Component	Entities without References
9	1	3	3	12	1
10	1	5	5	20	1
11	1	8	8	32	1
12	1	20	20	80	1
13	1	40	40	160	1
14	1	70	70	280	1
15	1	100	100	400	1

Table D.4: Measurement Result (Test 2)

Model #	CPU Time (s)	RAM (MB)	Generated Files	Package Size (MB)
9	0.76	4,30	110	0.76
10	1.17	6.80	160	1.10
11	1.89	10.20	235	1.70
12	5.77	17.40	535	3.90
13	14.12	34.40	1035	7.50
14	32.45	60.70	1785	14.00
15	55.47	87.00	2535	19.00

Scalability Test 3: Scaling the Number of Entities with References

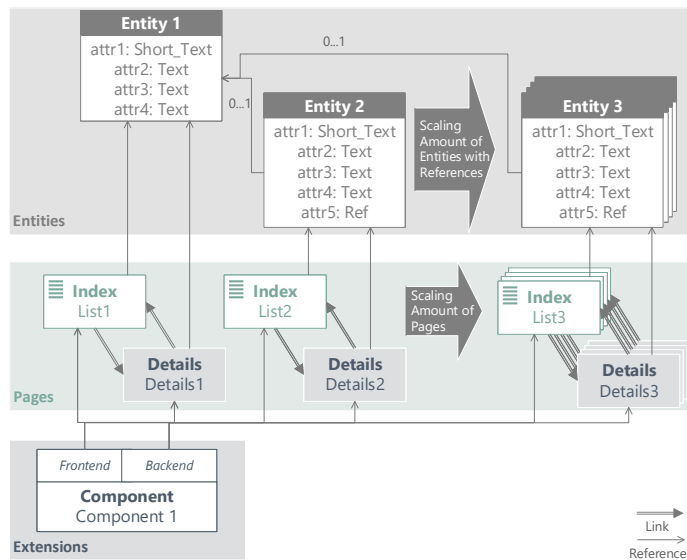


Figure D.3: Measurement Setting (Test 3)

Table D.5: Measurement Setting (Test 3)

Model #	Components	Index/Details Page Pairs	Referenced Pages per Component	Entities without References	Entities with References
16	1	2	8	1	1
17	1	4	16	1	3
18	1	6	24	1	5
19	1	9	36	1	8
20	1	21	84	1	20
21	1	41	164	1	40
22	1	71	284	1	70
23	1	101	404	1	100

Table D.6: Measurement Result (Test 3)

Model #	CPU Time (s)	RAM (MB)	Generated Files	Package Size (MB)
16	0.53	2	92	0.60
17	1.23	4.20	152	1.10
18	2.03	6.40	212	1.50
19	3.45	10.30	302	2.10
20	11.13	26.40	662	4.50
21	32.09	59.00	1262	8.70
22	79.91	122.00	2162	15.00
23	162.32	198.00	3062	21.00

Scalability Test 4: Scaling the Number of References in one Entity

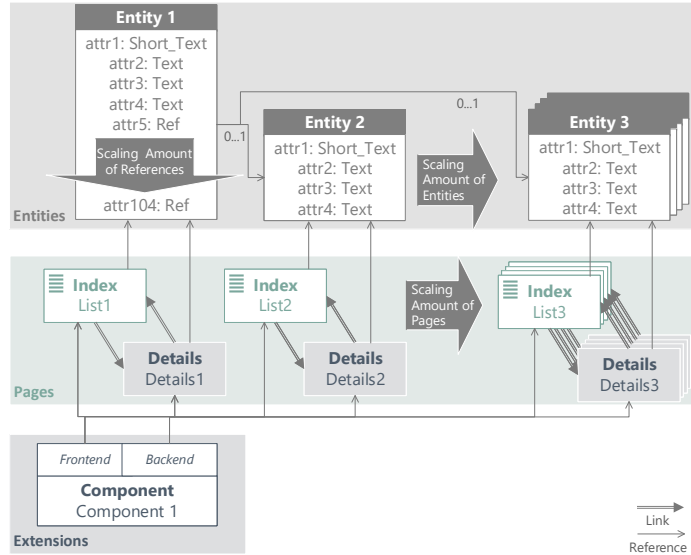


Figure D.4: Measurement Setting (Test 4)

Table D.7: Measurement Setting

Model #	Components	Index/Details Page Pairs	Referenced Pages per Component	Entities without References	Entities with References
24	1	4	16	3	1
25	1	6	24	5	1
26	1	9	36	8	1
27	1	21	84	20	1
28	1	41	164	40	1
29	1	71	284	70	1
30	1	101	404	100	1

Table D.8: Measurement Result (Test 4)

Model #	CPU Time (s)	RAM (MB)	Generated Files	Package Size (MB)
24	1.25	4.20	152	1.10
25	2.10	6.40	212	1.50
26	3.55	9.80	302	2.10
27	12.65	24.00	662	4.70
28	37.80	49.00	1262	9.00
29	102.93	91.70	2162	16.00
30	204.00	139.00	3062	22.00

Scalability Test 5: Scaling the Number of Entities with Many-to-Many References

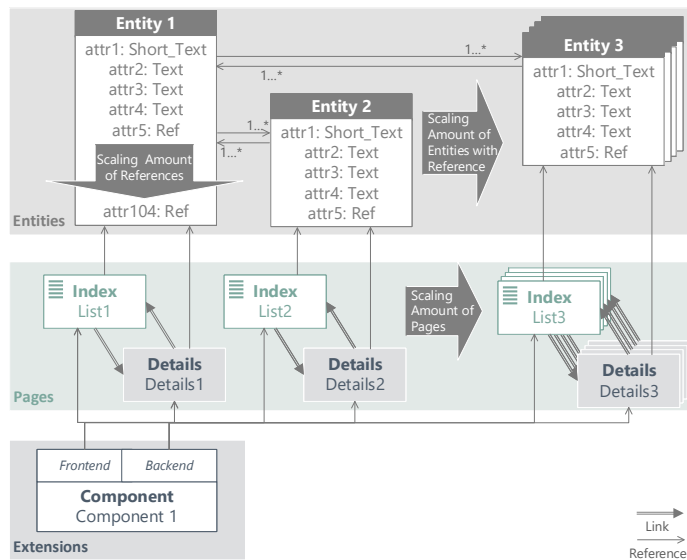


Figure D.5: Measurement Setting (Test 5)

Table D.9: Measurement Setting (Test 5)

Model #	Components	Index/Details Page Pairs	Referenced Pages per Component	References in Entity 1	Entities with Reference to Entity 1
31	1	2	8	1	1
32	1	4	16	3	3
33	1	6	24	5	5
34	1	9	36	8	8
35	1	21	84	20	20
36	1	41	164	40	40
37	1	71	284	70	70
38	1	101	404	100	100

Table D.10: Measurement Result (Test 5)

Model #	CPU Time (s)	RAM (MB)	Generated Files	Package Size (MB)
31	0.87	2.40	99	0.66
32	2.24	6.40	173	1.30
33	4.48	11.40	247	1.80
34	9.49	21.20	358	2.60
35	53.39	86.00	802	5.90
36	175.73	301.00	1542	12.00
37	618.00	895.00	2652	20.00
38	1110.00	1800.00	3762	29.00

E

Controlled Experiment: Documents

In this chapter, we present the artefacts of the controlled experiment (cf. Section 7.2). This includes the consent form, presentation, questionnaires, and tasks. Since the first iterations of the experiment have been conducted in Germany, the presentation and forms are in German language.

Experiment

Durchgeführt von:

- *Dennis Priefer*
- *Wolf Rost*

In diesem Experiment untersuchen wir den Einfluss einer MDD-Infrastruktur während der Umsetzung unterschiedlicher Anforderungen als Joomla- Erweiterungen.

Sie haben in diesem Experiment die Möglichkeit, ein neues Werkzeug zur modellgetriebenen Entwicklung von Joomla-Erweiterungen kennenzulernen und zu verwenden. Ihre gesamten Angaben erfolgen vollständig anonym. Persönliche Angaben erlauben keine Differenzierung oder Identifikation Ihrer Person, sondern dienen der üblichen statistischen Auswertung.

Einverständniserklärung

Hiermit erkläre ich, dass ich über Ziele und Ablauf der Studie informiert wurde und diese verstanden habe. Ich hatte ausreichend Gelegenheit, mich bei dem Versuchsleiter über die Studie zu informieren, sowie auftretende Fragen zu stellen. Diese wurden mir von der Versuchsleitung verständlich beantwortet. Mir ist bekannt, dass diese Studie in erster Linie der Wissenserweiterung dient und gegebenenfalls auch keinen persönlichen Vorteil für mich bringen kann.

Mit meiner Unterschrift erkläre ich, dass ich das Vorhaben und die Information verstanden habe und freiwillig an der Studie teilnehme. Ich habe verstanden, dass ich jederzeit ohne Angabe von Gründen aus der Studie ausscheiden kann, ohne dass mir persönliche Nachteile entstehen. Auch der Versuchsleiter oder die Versuchsleiterin kann die Studie jederzeit beenden.

Da keine personenbezogenen Daten erhoben werden, habe ich verstanden, dass nach Abschluss der Datenerhebung prinzipiell keine Zuordnung mehr zwischen den Daten im Datensatz und meinen personenbezogenen Daten möglich ist – der Datensatz ist anonym. Mir ist bewusst, dass nach Abschluss dieser Datenerhebung keine gezielte Löschung meines persönlichen Datensatzes mehr möglich ist, da dieser nicht zugeordnet werden kann. Eine Löschung aller erhobenen Originaldaten ist nach Ablauf der gesetzlichen Aufbewahrungsfrist von mindestens 10 Jahren vorgesehen.

Verwendung und Veröffentlichung der Daten: Ich habe verstanden, dass die Ergebnisse und Daten dieser Studie als wissenschaftliche Publikationen veröffentlicht werden und, dass dies in anonymisierter Form, d.h. ohne, dass die Daten einer spezifischen Person zugeordnet werden können, geschieht. Außerdem können die Datensätze nur von beteiligten Forschern der Studie eingesehen werden.

Zusendung der Ergebnisse

Ich möchte meine Ergebnisse (Fremdeinschätzung, Anforderung 1, Anforderung 2) elektronisch erhalten.

☐ nein

☐ ja, E-Mail Adresse: _____

Ort, Datum, Unterschrift

Figure E.1: Consent Form



Experiment

(Modellgetriebene) Entwicklung von Joomla-Erweiterungen

Durchgeführt von:

- Dennis Priefer (dennis.priefer@uni.thm.de)
- Wolf Rost (wolf.rost@uni.thm.de)

Einleitung

- Frage: Wie werden Joomla-Erweiterungen entwickelt?
 - Erfahrung egal
 - Vorgehen relevant
 - Wie schnell?
 - Qualität?
- Kann MDD die Entwicklung beschleunigen?
- Kann MDD die Qualität verbessern?


→ Experiment



Einleitung

Entwicklung eigener Erweiterungen für Joomla 3 in 2 Iterationen:

- Entwicklungsphase
 - Einrichtung IDE (nach Wahl), lokale Testinstanz
 - Anforderungsbeschreibung
 - Entwicklung → Vorgehen egal
 - Check, welche Akzeptanzkriterien erfüllt sind
- Entwicklungsphase
 - Web-Editor-Account nutzen, (neue) lokale Testinstanz (zurücksetzen)
 - Anforderungsbeschreibung
 - Entwicklung → Modellgetrieben mit JooMDD Web-Editor
 - Check, welche Akzeptanzkriterien erfüllt sind



Einleitung

Test-Instanzen werden bereitgestellt:
<https://icampus.thm.de/<experiment>>

Wir helfen nur bei Fragen zu:

- Joomla-Instanzen
- Anforderungen
- JooMDD

Definition of Done:

- Installierbar (zip-file)
- Multi-Language
- Joomla-Coding Standard
- auf Online-Server installiert und mit Testdaten befüllt
- CRUD (siehe Joomla Einführung)



Figure E.2: Experiment Presentation (1)

Einleitung

- Joomla-Einführung (ca. 10 Min)
- Umfrage (ca. 30 Min)
- 1. Entwicklungsphase (ca. 180 Min)
- Pause
- JooMDD-Einführung (ca. 25 Min)
- 2. Entwicklungsphase (ca. 180 Min)
- Abschluss (ca. 10 Min)



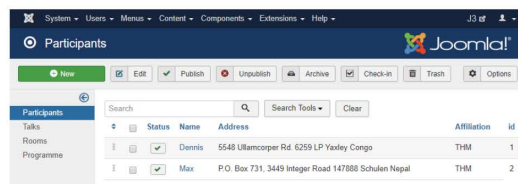
Umfrage

1. Fragen zur Person
2. Fragen zu Entwicklungserfahrung
3. Fragen zu Joomla-Kenntnissen
 - Selbsteinschätzung
 - Fremdeinschätzung
4. Fragen zu MDE-Kenntnissen
 - Selbsteinschätzung
 - Fremdeinschätzung



Joomla Einführung

Standard-Erweiterung: Listenansicht im Backend



Joomla Einführung

Standard-Erweiterung: Editieransicht im Backend

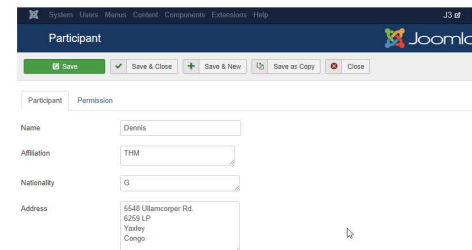
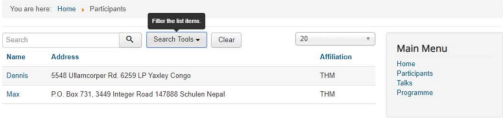


Figure E.3: Experiment Presentation (2)

Joomla Einführung


Standard-Erweiterung: Listenansicht im Frontend



The screenshot shows the Joomla! frontend interface for the 'Participants' component. It features a search bar at the top with a 'Search Tools' button and a 'Clear' button. Below the search bar is a table with columns 'Name', 'Address', and 'Affiliation'. The table contains two entries: 'Dennis' with address '5548 Ultracorpor Rd. 6259 LP Yaxley Congo' and affiliation 'THM', and 'Max' with address 'P.O. Box 731, 3449 Integer Road 147888 Schulen Nepal' and affiliation 'THM'. A 'Main Menu' sidebar on the right lists 'Home', 'Participants', 'Talks', and 'Programme'.

Joomla Einführung

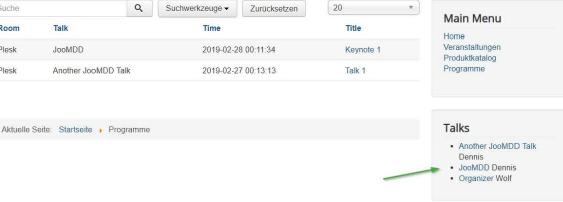
Standard-Erweiterung: Detailansicht im Frontend



The screenshot shows the Joomla! frontend interface for the 'Participants' component in detail view. It features a search bar at the top with a 'Search Tools' button and a 'Clear' button. Below the search bar is a form with fields for 'Name', 'Address', and 'Affiliation'. The form contains the following data: 'Name' is 'Dennis', 'Address' is '5548 Ultracorpor Rd. 6259 LP Yaxley Congo', and 'Affiliation' is 'THM'. A 'Main Menu' sidebar on the right lists 'Home', 'Participants', 'Talks', and 'Programme'.

Joomla Einführung

Standard-Erweiterung: Modul im Frontend

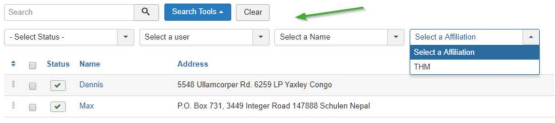


The screenshot shows the Joomla! frontend interface for the 'Participants' component in a module view. It features a search bar at the top with a 'Suchwerkzeuge' button and a 'Zurücksetzen' button. Below the search bar is a table with columns 'Room', 'Talk', 'Time', and 'Title'. The table contains two entries: 'Plesk' with talk 'JooMDD' and time '2019-02-28 00:11:34', and 'Plesk' with talk 'Another JooMDD Talk' and time '2019-02-27 00:13:13'. A 'Main Menu' sidebar on the right lists 'Home', 'Veranstaltungen', 'Produktkatalog', and 'Programme'. A 'Talks' sidebar on the left lists 'Another JooMDD Talk', 'JooMDD Dennis', and 'Organizer Wolf'.

Joomla-Einführung

Definition of Done - C(reate), R(ead), U(pdate), D(elete):

- Für alle Entitäten gilt, dass je eine Listenansicht sowie eine Editieransicht im Backend verfügbar sein muss.
- Die Listenansichten sollen durch Suchoptionen (Suche/Filter) anpassbar sein. Das heißt, die Suche, sowie Filter für alle Felder müssen implementiert werden.



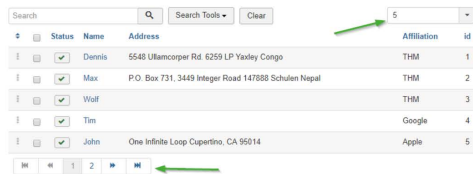
The screenshot shows the Joomla! frontend interface for the 'Participants' component in a list view. It features a search bar at the top with a 'Search Tools' button and a 'Clear' button. Below the search bar is a table with columns 'Status', 'Name', and 'Address'. The table contains two entries: 'Dennis' with address '5548 Ultracorpor Rd. 6259 LP Yaxley Congo' and 'Max' with address 'P.O. Box 731, 3449 Integer Road 147888 Schulen Nepal'. A 'Main Menu' sidebar on the right lists 'Home', 'Participants', 'Talks', and 'Programme'.

Figure E.4: Experiment Presentation (3)

Joomla-Einführung

Definition of Done - C(reate), R(ead), U(pdate), D(elete):

- Außerdem müssen die Listeneinträge über die Spaltenüberschriften sortierbar sein. Um lange Listen zu vermeiden, muss die Anzahl der darzustellenden Elemente auswählbar sein (Pagination). Dies gilt für Listenansichten im Backend und Frontend.



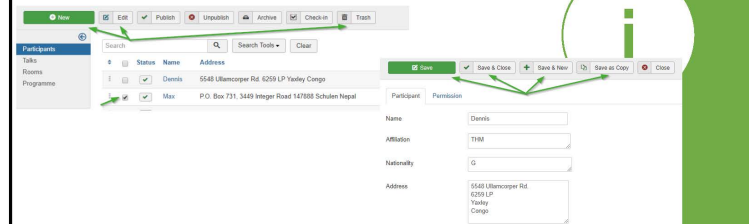
	Status	Name	Address	Affiliation	Id
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Dennis	5548 Ullamcorper Rd. 6259 LP Yaxley Congo	THM	1
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Max	P.O. Box 731, 3449 Integer Road 14788 Schulen Nepal	THM	2
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Wolf		THM	3
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Tim		Google	4
<input type="checkbox"/>	<input checked="" type="checkbox"/>	John	One Infinite Loop Cupertino, CA 95014	Apple	5

i

Joomla-Einführung

Definition of Done - C(reate), R(ead), U(pdate), D(elete):

- Die Ansichten im Backend müssen für alle Entitäten CRUD vollständig anbieten.



Participants

	Status	Name	Address	Affiliation	Id
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Dennis	5548 Ullamcorper Rd. 6259 LP Yaxley Congo	THM	1
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Max	P.O. Box 731, 3449 Integer Road 14788 Schulen Nepal	THM	2

Participant: Dennis

Name: Dennis

Affiliation: THM

Nationality: G

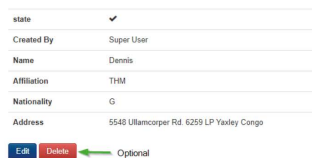
Address: 5548 Ullamcorper Rd. 6259 LP Yaxley Congo

i

Joomla-Einführung

Definition of Done - C(reate), R(ead), U(pdate), D(elete):

- Im Frontend werden Detailansichten grundsätzlich nicht zum Editieren verwendet - können aber (z.B. durch einen Editier-Button).



state: ☒

Created By: Super User

Name: Dennis

Affiliation: THM

Nationality: G

Address: 5548 Ullamcorper Rd. 6259 LP Yaxley Congo

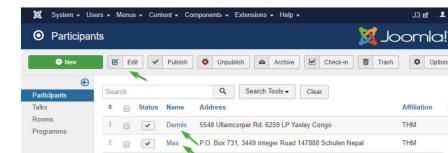
Optional

i

Joomla-Einführung

Definition of Done - C(reate), R(ead), U(pdate), D(elete):

- Detail-Ansichten können nur aus dem Kontext von Listenansichten geöffnet werden (z.B. neu oder bearbeiten von vorhandenem Listeneintrag).



Participants

	Status	Name	Address	Affiliation	Id
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Dennis	5548 Ullamcorper Rd. 6259 LP Yaxley Congo	THM	1
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Max	P.O. Box 731, 3449 Integer Road 14788 Schulen Nepal	THM	2

Figure E.5: Experiment Presentation (4)

Entwicklungsphase 1

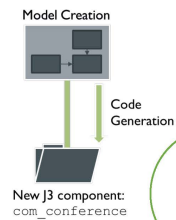
- Einrichtung
 - IDE (nach Wahl)
 - lokale Testinstanz
 - Bereitgestellte Testinstanzen:
<https://icampus.thm.de/experimentX>
- Entwicklung → Vorgehen egal
- Checkliste: Welche Anforderungen wurden realisiert?
- Kurze Umfrage zu Entwicklungsphase 1



Pause

JooMDD-Einführung

- Modellierungssprache
 - Daten-Modellierung
 - Interaction-Modellierung
 - Erweiterungs-Modellierung
 - <https://github.com/thm-mni-17/JooMDD/tree/master/docu>
- Code-Generator
 - Installierbare Erweiterungen für Joomla 3
- Webeditor
 - tinyurl.com/joomdd-web
 - Accounts vorhanden:
 Username: experiment<X>
 Passwort: Gleiches PW wie Joomla-Instanz



JooMDD-Einführung Webeditor

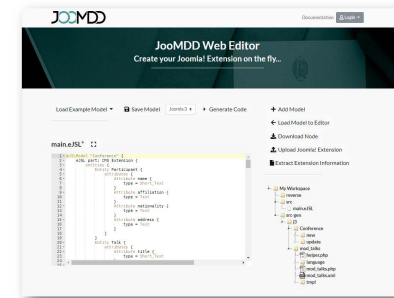


Figure E.6: Experiment Presentation (5)

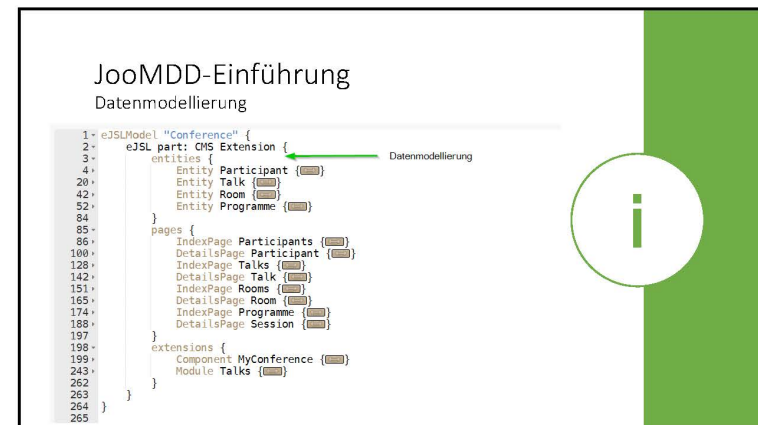
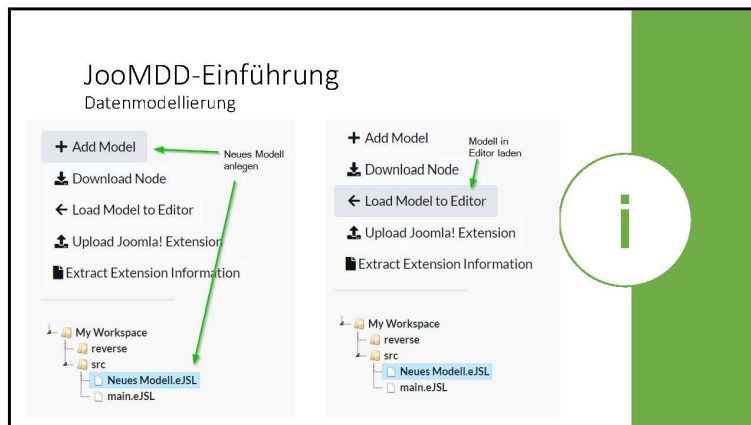
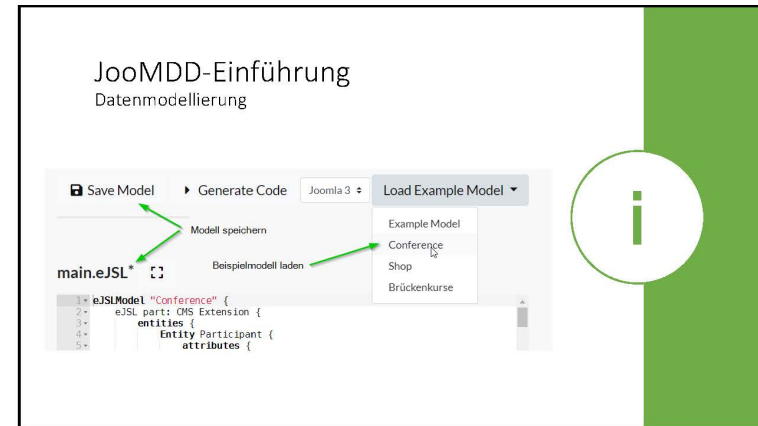
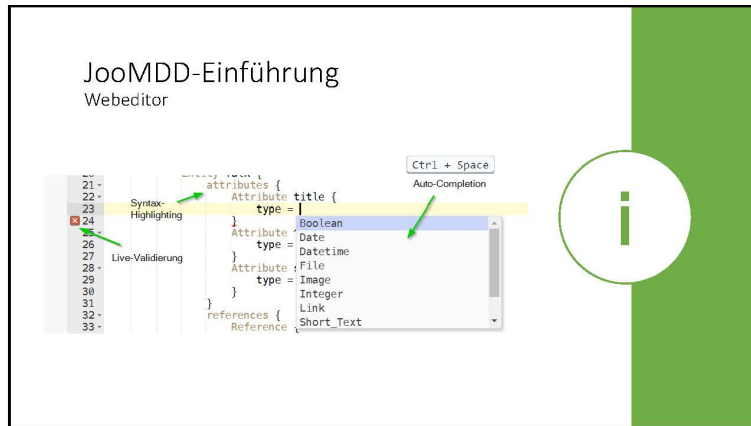


Figure E.7: Experiment Presentation (6)

JooMDD-Einführung

Datenmodellierung

```

Entity Participant {
  attributes {
    Attribute name {
      type = Short_Text
    }
    Attribute affiliation {
      type = Text
    }
    Attribute nationality {
      type = Text
    }
    Attribute address {
      type = Text
    }
  }
}

Entity Talk {
  attributes {
    Attribute title {
      type = Short_Text
    }
    Attribute description {
      type = Text
    }
    Attribute speaker {
      type = Short_Text
    }
  }
  references {
    EntityAttribute = speaker
    ReferencedEntity = Participant
    ReferencedEntityAttribute = Participant.name
    min = 1
    max = 1
  }
}

```

Attribut-Definitionen

Datenbank-Typen (abstrahiert)

Referenz zu anderer Entität

Multiplizität (1..1)

JooMDD-Einführung

Interaction-Modellierung

```

1= eJSLModel "Conference" {
2=   eJSL part: CMS Extension {
3=     entities {
4=       Entity Participant {
20=       Entity Talk {
42=       Entity Room {
52=       Entity Programme {
84=     }
85=     pages {
86=       IndexPage Participants {
100=       DetailsPage Participant {
128=       IndexPage Talks {
142=       DetailsPage Talk {
151=       IndexPage Rooms {
165=       DetailsPage Room {
174=       IndexPage Programme {
188=       DetailsPage Session {
197=     }
198=     extensions {
199=       Component MyConference {
243=       Module Talks {
262=     }
263=   }
264= }
265=

```

Interaction-Modellierung

JooMDD-Einführung

Interaction-Modellierung

```

IndexPage Participants {
  *Entities Participant
  representation columns = Participant.name, Participant.address,
  Participant.affiliation
  filters = Participant.name, Participant.affiliation
  links {
    InternalContextLink Details {
      target = Participant
      linked attribute = Participant.name
      linkparameters {
        Parameter name = *Attribute "Participant.name"
      }
    }
  }
}

DetailsPage Participant {
  *Entities Participant
  editfields {
    InternalLink Index {
      target = Participants
      linked attribute = name
    }
  }
}

```

Welche Entität soll dargestellt werden

Welche Filter

Welche Spalten-Überschriften

Link zu Detail/Editor-Ansicht

Kontext-Parameter

Welche Felder

Link zu Listenansicht (Muss immer gesetzt werden)

JooMDD-Einführung

Interaction-Modellierung

```

DetailsPage Participant {
  *Entities Participant
  editfields {
    Participant.name {
      html type = Text_Field
    },
    Participant.address {
      html type = Textarea
    },
    Participant.affiliation {
      html type = Editor
    },
    Participant.nationality {
      html type = Select
      values {
        en="English",
        de="German",
        other="Other"
      }
    }
  }
  links {
  }
}

```

* Optionale Angabe der Editor-felder

HTML-Typen

Select-Options

Figure E.8: Experiment Presentation (7)

JooMDD-Einführung

Erweiterungs-Modellierung

```

1- eJSLModel "Conference" {
2-   eJSL part: CMS Extension {
3-     entities {
4-       Entity Participant {}
5-       Entity Talk {}
6-       Entity Room {}
7-       Entity Programme {}
8-     }
9-     pages {
10-      IndexPage Participants {}
11-      DetailsPage Participant {}
12-      IndexPage Talks {}
13-      DetailsPage Talk {}
14-      IndexPage Rooms {}
15-      DetailsPage Room {}
16-      IndexPage Programme {}
17-      DetailsPage Session {}
18-    }
19-     extensions {
20-       Component MyConference {}
21-       Module Talks {}
22-     }
23-   }
24- }

```

Erweiterungs-Modellierung

JooMDD-Einführung

Erweiterungs-Modellierung

```

Component MyConference {
  Manifestation {
    authors {}
    copyright = "Copyright (C) 2019 All right reserved."
    license = "GNU General Public License"
    version = "1.0.0"
  }
  languages {
    Language de-DE {}
    Language en-GB {}
    Language system de-DE {}
    Language system en-GB {}
  }
  sections {}
}

```

Meta-Daten

Welche Sprachdateien (Default: en-GB) (eigene Konstanten können definiert werden, müssen aber nicht)

MYCONFERENCE_HELLO_WORLD = "Hallo Welt"

JooMDD-Einführung

Erweiterungs-Modellierung: Joomla-Komponente

```

Component MyConference {
  Manifestation {}
  languages {}
  sections {
    Frontend section {}
    Backend section {}
  }
  *Pages {
    *Page : Participants
    *Page : Participant
    *Page : Talks
    *Page : Talk
    *Page : Rooms
    *Page : Room
    *Page : Programme
    *Page : Session
  }
}

```

Welche Page soll für Frontend/ Backend verwendet werden (MVC-Dateien)

JooMDD-Einführung

Erweiterungs-Modellierung: Joomla-Modul

```

Module Talks {
  Manifestation {
    authors {
      Author "John Doe" {
        authoremail = "john.doe@example.org"
      }
    }
    copyright = "Copyright (C) 2019 All right reserved."
    license = "GNU General Public License"
    version = "1.0.0"
  }
  languages {
    Language de-DE {}
    Language system de-DE {}
    Language en-GB {}
    Language system en-GB {}
  }
  *Page : Talks from : MyConference data : backendDAO : backendDAO : database : frontendDAO : webservice
}

```

Welche Page soll dargestellt werden

Woher kommen die Daten (Im Beispiel: Model der Backend Talks-View der MyConference-Komponente)

Talks

- Another JooMDD Talk
- Dennis
- JooMDD Dennis
- Organizer Wolf

Figure E.9: Experiment Presentation (8)

JooMDD-Einführung

Codegenerierung

My Workspace

- reverse
- src
 - Neues Modell.eJSL
 - main.eJSL
- src-gen
- j3
 - Conference
 - new
 - com_myconference
 - administrator
 - com_myconference.xt
 - components
 - media
 - script.php
 - update
 - mod_talks
 - Download
 - Remove

Rechtsklick: Download als zip

Success! Code successful generated.

Entwicklungsphase 2

- Einrichtung
 - Account in Webeditor nutzen
 - (Neue lokale Joomla-Instanz)
- Entwicklung → Anforderung mit JooMDD umsetzen
- Check: Welche Anforderungen wurden realisiert?
- Kurze Umfrage zu Entwicklungsphase 2

Abschluss

- Feedback zu Entwicklungsphasen

Figure E.10: Experiment Presentation (9)

Umfrage:

Datensatz-ID:

Vielen Dank für Ihre Teilnahme an unserem Experiment. Damit wir die Ergebnisse des Experiments besser auswerten können, benötigen wir noch einige Informationen. Diese helfen uns dabei, den aktuellen Stand unserer Werkzeuge zu bewerten und diese damit zu verbessern.

Die 1. Umfrage ist in 4 Teile aufgebaut. Im 1. Teil möchten wir einige Informationen zu Ihrer Person sammeln, selbstverständlich in anonymisierter Form. Im 2. Teil möchten wir etwas über Ihre Erfahrung im Bereich der Softwareentwicklung erfahren. Der 3. Teil besteht aus einer Selbst- und Fremdeinschätzung zu Ihren Joomla-Kenntnissen. Im 4. Teil möchten wir Ihre MDE-Kenntnisse ermitteln.

Bitte beachten Sie folgende Konventionen zur Auswahl Ihrer Antwortmöglichkeiten:

- ☐ : Bitte kreuzen Sie nur **eine** Antwortmöglichkeit an
- ☐ : Bitte kreuzen Sie **eine oder mehrere** Antwortmöglichkeiten an
- : Bitte tragen Sie **eine Zahl** ein

Wir bitten Sie die jeweiligen Abschnitte gewissenhaft und sorgfältig zu bearbeiten. Bei Fragen können Sie sich jederzeit an die anwesenden Versuchsleiter wenden.

Sie können sich auf Wunsch die Ergebnisse des Experiments zusenden lassen. Hierzu erhalten Sie am Ende des Fragebogens eine ID, welche Ihnen die Zuordnung Ihres Datensatzes ermöglicht.

Teil 1: Fragen zur Person

1.1 Alter

- ☐ keine Angabe
- ☐ < 20 ☐ 20-25 ☐ 26-30 ☐ > 30

1.2 Geschlecht

- ☐ keine Angabe
- ☐ männlich ☐ weiblich ☐ divers

1.3 Deutschkenntnisse (Schulnoten)

- Keine ☐ 6 ☐ 5 ☐ 4 ☐ 3 ☐ 2 ☐ 1 Sehr gut

1.4 Englischkenntnisse (Schulnoten)

- Keine ☐ 6 ☐ 5 ☐ 4 ☐ 3 ☐ 2 ☐ 1 Sehr gut

Figure E.11: Demographic Questionnaire

Teil 2: Fragen zu Entwicklungserfahrung

2.1 Wie hoch schätzen Sie Ihre Erfahrung als Software-Entwickler ein?

Unerfahren ☐ 1 ☐ 2 ☐ 3 ☐ 4 Sehr erfahren

2.2 Wie hoch schätzen Sie Ihre Kenntnisse bei der Implementierung von Anforderungen in einer Technologie aus dem Web-Bereich (z.B. PHP, JS, HTML/JS etc.) ein?

☐ keine ☐ niedrig ☐ ausreichend ☐ befriedigend ☐ hoch ☐ sehr hoch

2.3 Wie viel Erfahrung haben Sie als Softwareentwickler (in Jahren)?

☐ < 1 ☐ 2-5 ☐ 6-10 ☐ 11-20 ☐ > 20

2.4 Haben Sie bisher schon mit Entwicklungsumgebungen gearbeitet?

☐ nein ☐ ja

2.4.1 Wenn ja, welche Entwicklungsumgebungen benutzen Sie regelmäßig?

- ☐ Eclipse
- ☐ JetBrains
- ☐ Cloud-IDE (Web-Editor)
- ☐ Text-Editor, welchen: _____
- Andere, _____

2.5 Haben Sie bereits vor dem Experiment aus Anforderungen oder Anforderungsteilen eine Implementierung erstellt?

☐ nein ☐ ja

Teil 3a: Fragen zu Joomla-Kenntnissen (Selbsteinschätzung)

3.1 Haben Sie bereits vor dem Experiment mit einem CMS wie Joomla, WordPress, o.ä. gearbeitet?

☐ nein ☐ ja

Wenn ja, als

- ☐ Benutzer
- ☐ Administrator
- ☐ Entwickler

3.2 Wie viel Erfahrung haben Sie im Umgang mit einem CMS (in Jahren)?

☐ keine ☐ < 1 ☐ 2-5 ☐ 6-10 ☐ 11-20 ☐ > 20

3.3 Haben Sie bereits vor dem Experiment Software-Erweiterungen für Joomla realisiert?

☐ nein ☐ ja

3.3.1 Wenn ja, zählen dazu auch Erweiterungen, welche auf vorhandenen 3rd-Party Erweiterungen aufbauen oder diese verwenden (z.B. deren Daten)?

☐ nein ☐ ja

3.4 Wie viel Erfahrung haben Sie als Entwickler von Joomla-Erweiterungen (in Jahren)?

☐ keine ☐ < 1 ☐ 2-5 ☐ 6-10 ☐ 11-20

Figure E.12: Self-Assessment (Self-Assessment: General Software Development and Joomla)

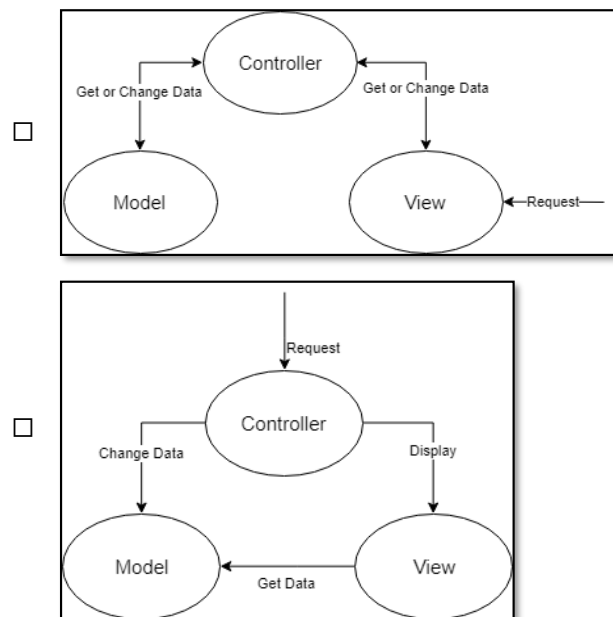
Teil 3.2: Fragen zu Joomla-Kenntnissen (Fremdeinschätzung)

3.5 Welche Erweiterungstypen gibt es in Joomla? Kreuzen Sie die Typen an, die Ihnen bekannt sind?

- ☐ Plugin ☐ Modul ☐ App ☐ Segment ☐ Bibliothek ☐ Komponente ☐ Widget
☐ Template ☐ Injection ☐ Wrap

Andere, _____

3.6 Kreuzen Sie alle richtigen Diagramme an, welche die üblichen MVC-Muster in Komponenten abbilden.



3.7 Kreuzen Sie die richtigen Aussagen an:

- ☐ Im Frontend kann der Administrator die Struktur der Webseite verwalten.
☐ Im Backend kann der Administrator die Struktur der Webseite verwalten.
☐ Eine Komponente kann aus einem Frontend- sowie einem Backend-Anteil bestehen.
☐ Eine Komponente hat immer nur einen Frontend-Anteil.
☐ Eine Komponente hat immer nur einen Backend-Anteil.
☐ Ein Modul kann auf die Daten einer Komponente zugreifen.
☐ Ein Modul kann nur eigene Daten darstellen.
☐ Ein Manifest ist zwingend erforderlich, um eine Erweiterung zu installieren.
☐ Ein Manifest ist optional.
☐ Ein Manifest enthält wichtige Metadaten der Erweiterung.
☐ Ein Manifest enthält SQL-Abfragen.
☐ Joomla erlaubt Mehrsprachigkeit durch die Verwendung der Datenbank.
☐ Mehrsprachigkeit wird durch Sprachdateien (Key-Value) realisiert.
☐ Mehrsprachigkeit wird durch Sprach-Erweiterungen realisiert.
☐ Joomla unterstützt keine Mehrsprachigkeit.
☐ Es gibt unterschiedliche Typen von Plugins, z.B. Such-, System- und Userplugins.
☐ Plugins werden immer von den gleichen Events ausgelöst.

Figure E.13: External Assessment: Joomla Knowledge)

Teil 4.1: Fragen zu MDE-Kenntnissen (Selbsteinschätzung)

4.1 Haben Sie bereits vor dem Experiment mit Modellierungssprachen gearbeitet?

☐ nein ☐ ja

Wenn ja, mit

☐ UML ☐ EMF Ecore ☐ BPMN ☐ BPEL ☐ LabView

☐ andere, _____

4.2 Wie hoch schätzen Sie Ihre Kenntnisse bei der Modellierung von Anforderungen in einer Modellierungssprache (z.B. UML; EMF Ecore, BPMN, BPEL, etc.) ein?

☐ keine ☐ niedrig ☐ ausreichend ☐ befriedigend ☐ hoch ☐ sehr hoch

4.3 Wie viel Erfahrung haben Sie im Umgang mit Modellierungssprachen (in Jahren)?

☐ keine ☐ < 1 ☐ 2-5 ☐ 6-10 ☐ 11-20 ☐ > 20

4.4 Wie groß schätzen Sie den Bedarf eines Werkzeugs/Methode zur Formulierung und Überführung von Anforderungsdokumenten in Modelle und Implementierungen ein?

☐ kein Bedarf ☐ niedrig ☐ mittel ☐ hoch ☐ sehr hoch

4.5 Haben Sie bereits vor dem Experiment Codegeneratoren verwendet (z.B. zur Generierung von Joomla-Erweiterungen)?

☐ nein ☐ ja

Wenn ja, zur

☐ Generierung von Dokumentation

☐ Generierung von Joomla-Erweiterungen

☐ Generierung von mobilen Anwendungen

☐ Andere, _____

Figure E.14: Self-Assessment (Self-Assessment: MDE)

Teil 4.2: Fragen zu MDE-Kenntnissen (Fremdeinschätzung)

4.6 Ergänzen Sie die Implementierung mit Hilfe der möglichen Lösungen anhand der Anforderung (durch Mehrfachzuordnung der Nummern).

<p>Eine Konferenz bietet Vorträge an. Diese können von Besuchern der Konferenz besucht werden. Besucher haben eine Teilnehmernummer. Jeder Mitarbeiter des Konferenzteams hat eine Personalnummer und genau einen Mitarbeiter als Vorgesetzten. Besucher und Mitarbeiter haben einen Namen.</p>	<p>1) Person 2) Mitarbeiter 3) Besucher 4) List<Vortrag> 5) Vortrag 6) vorgesetzter</p>	<pre> public class ○ { private String name; } public class ○ extends ○ { private int teilnehmerNummer; private ○ besucht; } public class ○ extends ○ { private int personalNummer; private ○ ○ ; } public class Konferenz { private List<Person> personen; private ○ vortraege; } public class ○ {} </pre>
---	---	--

4.7 Ergänzen Sie das Modell mit Hilfe der möglichen Lösungen anhand der Anforderung (durch Mehrfachzuordnung der Nummern).

<p>Ein Krankenhaus besitzt ein oder mehrere Stationen. Eine Station gehört genau einem Krankenhaus an. Die Stationen können mit Patienten belegt sein. Jeder Patient ist jedoch genau einer Station zugeordnet. Das Krankenhaus beschäftigt mindestens ein Team für die Versorgung der Patienten. Ein Team arbeitet nur für ein Krankenhaus. Ein Team besteht aus einem oder mehreren Ärzten. Ein Arzt muss keinem Team angehören. Jedes Team wird von genau einem Arzt geleitet, während es Ärzte ohne Leitungsfunktion gibt. Ein Arzt betreut einen oder mehrere Patienten. Jeder Patient wird von mehreren Ärzten, mindestens jedoch einem Arzt behandelt.</p>	
<p>1. [0..1] 2. [1..1] 3. [0..*]</p>	<p>4. [1..*] 5. \longrightarrow (Assoziation) 6. \blacklozenge (Komposition)</p>

Figure E.15: Self-Assessment (External Assessment: MDE)

Umfrage zu Entwicklungsphase 1:

Datensatz-ID:

5.1 Wie sind Sie vorgegangen?

- ☐ From Scratch (z.b. Tutorial, Doku)
- ☐ Clone-and-own (copy&paste)
- ☐ Orientiert an existierenden Erweiterungen
 - ☐ Eigene
 - ☐ 3rd-party
 - ☐ Core
- ☐ Generator, welcher _____

5.2 Konnten alle Anforderungen umgesetzt werden?

- ☐ keine (0) ☐ nein (mindestens eine nicht umgesetzt) ☐ ja (alle)

5.3 Haben Sie eine oder mehrere Entwicklungsumgebungen verwendet?

- ☐ nein ☐ ja

5.3.1 Wenn ja, welche?

- ☐ Eclipse
- ☐ JetBrains
- ☐ Cloud-IDE (Web-Editor)
- ☐ Text-Editor (Welcher)
- ☐ Andere, _____

5.4 Haben Sie schon einmal ein Projekt mit vergleichbarem Umfang umgesetzt?

- ☐ nein ☐ ja

Figure E.16: Questionnaire after Session 1

Umfrage zu Entwicklungsphase 2:

Datensatz-ID:

6.1 Konnten alle Anforderungen umgesetzt werden?

- ☐ keine ☐ nein ☐ ja

6.2 Konnten Sie alle Anforderungen mit der gegebenen Modellierungssprache abbilden?

- ☐ nein ☐ ja

6.3 Konnten Sie aus Ihrem Modell installierbare Joomla-Erweiterungen generieren?

- ☐ nein ☐ ja

6.3.1 Wenn ja, mussten Sie den generierten Code von Hand anpassen?

- ☐ nein ☐ ja

Figure E.17: Questionnaire after Session 2

Feedback:

Datensatz-ID:

7.1 Hatten Sie ausreichend Zeit für die gestellten Aufgaben?

☐ zu wenig ☐ passend ☐ zu viel

7.2 Konnten Sie durch das Verwenden der gegebenen Werkzeuge zur modellgetriebenen Entwicklung der Erweiterungen mehr Anforderungen umsetzen?

☐ nein ☐ ja

7.3 Fanden Sie die Entwicklung mit den gegebenen Werkzeugen zur modellgetriebenen Entwicklung der Erweiterungen komfortabler im Vergleich zu Ihrer konventionellen Entwicklungsmethode?

☐ nein ☐ ja

7.4 Konnten Sie durch die Verwendung der gegebenen Werkzeuge zur modellgetriebenen Entwicklung der Erweiterungen die Entwicklung beschleunigen?

☐ nein ☐ ja

7.5 Hatten Sie das Gefühl, dass die Experiment-Umgebung Ihre Entwicklung beeinflusst hat?

☐ nein ☐ ja

Wenn ja

☐ Bei der Umsetzung von Anforderung 1

☐ Bei der Umsetzung von Anforderung 2

7.6 Was wollen Sie uns sonst noch mitteilen?

Figure E.18: Feedback Questionnaire

Anforderung A

Erstellen Sie eine **Joomla-Komponente** zur Verwaltung von Hochschulveranstaltungen (Modulhandbuch). Berücksichtigen Sie dabei die folgenden **CRUD-Akzeptanzkriterien (Definition of Done)**:

- *Für alle Entitäten gilt, dass je eine Listenansicht sowie eine Editieransicht im Backend verfügbar sein muss.*
- *Die Listenansicht sollen durch Suchoptionen (Suche/Filter) anpassbar sein. Das heißt, die Suche, sowie Filter für alle Felder müssen implementiert werden.*
- *Außerdem müssen die Listeneinträge über die Spaltenüberschriften sortierbar sein. Um lange Listen zu vermeiden, muss die Anzahl der darzustellenden Elemente auswählbar sein (Pagination). Dies gilt für Listenansichten im Backend und Frontend.*
- *Die Ansichten im Backend müssen für alle Entitäten CRUD vollständig anbieten. Im Frontend werden Detailansichten grundsätzlich nicht zum Editieren verwendet - können aber (z.B. durch einen Editier-Button).*
- *Detailansichten und Editieransichten können nur aus dem Kontext von Listenansichten geöffnet werden (z.B. neu oder bearbeiten von vorhandenem Listeneintrag).*

Eine Hochschule hat eine oder mehrere Niederlassungen und beheimatet in der Regel Fachbereiche, welche einer Niederlassung zugeordnet und wiederum unterschiedliche Studiengänge anbieten. Fachbereiche haben einen Namen, sowie eine Fachbereichsnummer. Niederlassungen werden durch einen Ort und eine Adresse definiert.

Studiengänge werden durch einen Namen und des zu erlangenden Abschlusses (B.A., B.Sc., B.Eng, M.A., M.Sc., M.Eng.), sowie durch die Anzahl der benötigten Semester beschrieben.

Das Angebot an Veranstaltungen der Hochschule umfasst Studiengangs- und sogar Fachbereichs-übergreifende Veranstaltungen. Veranstaltungen haben je eine englische und deutsche Bezeichnung, eine Modulnummer, eine Beschreibung und einen Typ (vorgegeben: Vorlesung, Seminar, Praktikum). Außerdem muss die Anzahl an Leistungspunkten (CrP) vergeben werden.

Für jede Veranstaltung muss ein verantwortlicher Professor festgelegt werden. Neben üblichen Merkmalen (Name, Adresse, Geburtsdatum) werden Professoren genau einem Fachgebiet (z.B. Machine Learning) zugeordnet, welches ebenfalls verwaltet werden kann (Bezeichnung des jeweiligen Fachgebietes ausreichend). Außerdem gehören Professoren immer genau einem Fachbereich an.

Im Frontend bietet das Modulhandbuch eine Übersicht über alle Veranstaltungen eines Studiengangs. In diesem werden der deutsche Veranstaltungstitel und der verantwortliche Professor für jede Veranstaltung aufgelistet. Durch die Eingabe von Suchkriterien lässt sich die Liste nach Veranstaltungstitel und Professor filtern. Der Veranstaltungstitel führt beim Anklicken zu einer detaillierteren Ansicht, in welcher zusätzlich noch die Voraussetzungen (andere Veranstaltungen) und Leistungspunkte angezeigt werden.

Erstellen Sie neben der Komponente ein **Joomla-Modul**, welches alle Professoren (Name) jeweils mit ihrem zugeordneten Fachbereich als Liste darstellt.

Figure E.19: Requirement A: University Management

Akzeptanzkriterien - Checkliste:

Struktur der Erweiterungen:

- ☐ Ist die Komponente installierbar?
- ☐ Ist ein Update-Skript implementiert?
- ☐ Ist die Mehrsprachigkeit (durch Sprachkonstanten und -dateien) realisiert?
- ☐ Ist das Modul implementiert?
- ☐ Ist das Modul installierbar?
- ☐ Stellt das Modul die gewünschten Daten der Komponente dar?

Ansichten für **Niederlassungen**:

- ☐ Gibt es eine Listenansicht für Niederlassungen im Backend?
Wenn ja, kreuzen Sie die vorhandenen Tabellenüberschriften an:
 - ☐ ID ☐ Ort ☐ Adresse
 - ☐ Können die Listeneinträge über die Spaltenüberschriften sortiert werden?
 - ☐ Gibt es Such- und Filteroptionen für die Listeneinträge?
- ☐ Gibt es eine Editieransicht für Niederlassungen im Backend?
Wenn ja, kreuzen Sie die vorhandenen Felder an:
 - ☐ Ort ☐ Adresse
- ☐ Gibt es eine Verlinkung von der Liste aller Niederlassungen zur Editieransicht einer Niederlassung im Backend?

CRUD-Funktionalität für **Niederlassungen**:

- ☐ Gibt es Buttons zum Erstellen, Editieren und Löschen in der Listenansicht von Niederlassungen im Backend?
Wenn ja,
 - ☐ Erstellen: Wird zu einer leeren Editieransicht zum Anlegen einer Niederlassung verlinkt?
 - ☐ Editieren: Wird zu einer Editieransicht zum Editieren einer vorhandenen Niederlassung verlinkt?
 - ☐ Editieren: Sind die Werte der ausgewählten Niederlassung bereits in editierbaren Feldern dargestellt?
 - ☐ Löschen: Werden einzeln ausgewählte Einträge erfolgreich gelöscht?
 - ☐ Löschen: Können mehrere Einträge auf einmal gelöscht werden?
- ☐ Gibt es einen "Speichern"-Button in der Editieransicht einer Niederlassung im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und wieder die gleiche Editieransicht angezeigt?
- ☐ Gibt es einen "Speichern & Schließen"-Button in der Editieransicht einer Niederlassung im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und die zugehörige Listenansicht angezeigt?
- ☐ Gibt es einen "Speichern & Neu"-Button in der Editieransicht einer Niederlassung im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und eine leere Editieransicht angezeigt?
- ☐ Gibt es einen "Schließen"-Button in der Editieransicht einer Niederlassung im Backend?
 - ☐ Wenn ja, wird ohne zu speichern die zugehörige Listenansicht angezeigt?

Ansichten für **Fachbereiche**:

- ☐ Gibt es eine Listenansicht für Fachbereiche im Backend?
Wenn ja, kreuzen Sie die vorhandenen Tabellenüberschriften an:
 - ☐ ID ☐ Name ☐ Fachbereichsnummer ☐ Niederlassung
 - ☐ Gibt es eine Verlinkung zur Editieransicht der jeweiligen Niederlassungen in der Listenansicht?
 - ☐ Können die Listeneinträge über die Spaltenüberschriften sortiert werden?
 - ☐ Gibt es Such- und Filteroptionen für die Listeneinträge?
- ☐ Gibt es eine Editieransicht für Fachbereiche im Backend?
Wenn ja, kreuzen Sie die vorhandenen Felder an:
 - ☐ Name ☐ Fachbereichsnummer
 - ☐ Gibt es ein Feld "Niederlassung" o.ä.?
 - ☐ Wenn ja, kann aus einer Liste vorhandener Niederlassungen gewählt werden?
- ☐ Gibt es eine Verlinkung von der Liste aller Fachbereiche zur Editieransicht eines Fachbereichs im Backend?

CRUD-Funktionalität für **Fachbereiche**:

- ☐ Gibt es Buttons zum Erstellen, Editieren und Löschen in der Listenansicht von Fachbereichen im Backend?
Wenn ja,
 - ☐ Erstellen: Wird zu einer leeren Editieransicht zum Anlegen eines Fachbereichs verlinkt?
 - ☐ Editieren: Wird zu einer Editieransicht zum Editieren eines vorhandenen Fachbereichs verlinkt?
 - ☐ Editieren: Sind die Werte des ausgewählten Fachbereichs bereits in editierbaren Feldern dargestellt?
 - ☐ Löschen: Werden einzeln ausgewählte Einträge erfolgreich gelöscht?
 - ☐ Löschen: Können mehrere Einträge auf einmal gelöscht werden?
- ☐ Gibt es einen "Speichern"-Button in der Editieransicht eines Fachbereichs im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und wieder die gleiche Editieransicht angezeigt?
- ☐ Gibt es einen "Speichern & Schließen"-Button in der Editieransicht eines Fachbereichs im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und die zugehörige Listenansicht angezeigt?
- ☐ Gibt es einen "Speichern & Neu"-Button in der Editieransicht eines Fachbereichs im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und eine leere Editieransicht angezeigt?
- ☐ Gibt es einen "Schließen"-Button in der Editieransicht eines Fachbereichs im Backend?
 - ☐ Wenn ja, wird ohne zu speichern die zugehörige Listenansicht angezeigt?

Figure E.20: Requirement A: Test Cases (1)

Ansichten für Studiengänge:

- ☐ Gibt es eine Listenansicht für Studiengänge im Backend?
Wenn ja, kreuzen Sie die vorhandenen Tabellenüberschriften an:
☐ ID ☐ Name ☐ Abschluss ☐ Anzahl Semester ☐ Fachbereich
☐ Gibt es eine Verlinkung zur Editieransicht der jeweiligen Fachbereiche in der Listenansicht?
☐ Können die Listeneinträge über die Spaltenüberschriften sortiert werden?
☐ Gibt es Such- und Filteroptionen für die Listeneinträge?
- ☐ Gibt es eine Editieransicht für Studiengänge im Backend?
Wenn ja, kreuzen Sie die vorhandenen Felder an:
☐ Name ☐ Anzahl Semester
☐ Gibt es ein Feld "Abschluss" o.ä.?
☐ Wenn ja, kann aus einer Liste vorhandener möglicher Abschlüsse gewählt werden?
☐ Gibt es ein Feld "Fachbereich" o.ä.?
☐ Wenn ja, kann aus einer Liste vorhandener Niederlassungen gewählt werden?
☐ Gibt es ein Feld "Veranstaltungen" o.ä.?
☐ Wenn ja, können beliebig viele Elemente aus einer Liste vorhandener Niederlassungen gewählt werden?
- ☐ Gibt es eine Verlinkung von der Liste aller Studiengänge zur Editieransicht eines Studiengangs im Backend?

CRUD-Funktionalität für Studiengänge:

- ☐ Gibt es Buttons zum Erstellen, Editieren und Löschen in der Listenansicht von Studiengängen im Backend?
Wenn ja,
☐ Erstellen: Wird zu einer leeren Editieransicht zum Anlegen eines Studiengangs verlinkt?
☐ Editieren: Wird zu einer Editieransicht zum Editieren eines vorhandenen Studiengangs verlinkt?
☐ Editieren: Sind die Werte des ausgewählten Studiengangs bereits in editierbaren Feldern dargestellt?
☐ Löschen: Werden einzeln ausgewählte Einträge erfolgreich gelöscht?
☐ Löschen: Können mehrere Einträge auf einmal gelöscht werden?
- ☐ Gibt es einen "Speichern"-Button in der Editieransicht eines Studiengangs im Backend?
☐ Wenn ja, werden die geänderten Daten gespeichert und wieder die gleiche Editieransicht angezeigt?
- ☐ Gibt es einen "Speichern & Schließen"-Button in der Editieransicht eines Studiengangs im Backend?
☐ Wenn ja, werden die geänderten Daten gespeichert und die zugehörige Listenansicht angezeigt?
- ☐ Gibt es einen "Speichern & Neu"-Button in der Editieransicht eines Studiengangs im Backend?
☐ Wenn ja, werden die geänderten Daten gespeichert und eine leere Editieransicht angezeigt?
- ☐ Gibt es einen "Schließen"-Button in der Editieransicht eines Studiengangs im Backend?
☐ Wenn ja, wird ohne zu speichern die zugehörige Listenansicht angezeigt?

Ansichten für Veranstaltungen im Backend:

- ☐ Gibt es eine Listenansicht für Veranstaltungen im Backend?
Wenn ja, kreuzen Sie die vorhandenen Tabellenüberschriften an:
☐ ID ☐ Bezeichnung (eng.) ☐ Bezeichnung (dt.) ☐ Modulnummer
☐ Typ ☐ Leistungspunkte ☐ Verantwortlicher Professor
☐ Gibt es eine Verlinkung zur Editieransicht der jeweiligen Professoren in der Listenansicht?
☐ Können die Listeneinträge über die Spaltenüberschriften sortiert werden?
☐ Gibt es Such- und Filteroptionen für die Listeneinträge?
- ☐ Gibt es eine Editieransicht für Veranstaltungen im Backend?
Wenn ja, kreuzen Sie die vorhandenen Felder an:
☐ Bezeichnung (eng.) ☐ Bezeichnung (dt.) ☐ Beschreibung
☐ Modulnummer ☐ Leistungspunkte
☐ Gibt es ein Feld "Typ" o.ä.?
☐ Wenn ja, kann er aus einer Liste vorhandener Typen gewählt werden?
☐ Gibt es ein Feld "Verantwortlicher Professor" o.ä.?
☐ Wenn ja, kann er aus einer Liste vorhandener Professoren gewählt werden?
☐ Gibt es ein Feld "Studiengänge" o.ä.?
☐ Wenn ja, können beliebig viele Elemente aus einer Liste vorhandener Studiengänge ausgewählt werden?
- ☐ Gibt es eine Verlinkung von der Liste aller Veranstaltungen zur Editieransicht einer Veranstaltung im Backend?

CRUD-Funktionalität für Veranstaltungen im Backend:

- ☐ Gibt es Buttons zum Erstellen, Editieren und Löschen in der Listenansicht von Veranstaltungen im Backend?
Wenn ja,
☐ Erstellen: Wird zu einer leeren Editieransicht zum Anlegen einer Veranstaltung verlinkt?
☐ Editieren: Wird zu einer Editieransicht zum Editieren einer vorhandenen Veranstaltung verlinkt?
☐ Editieren: Sind die Werte der ausgewählten Veranstaltung bereits in editierbaren Feldern dargestellt?
☐ Löschen: Werden einzeln ausgewählte Einträge erfolgreich gelöscht?
☐ Löschen: Können mehrere Einträge auf einmal gelöscht werden?
- ☐ Gibt es einen "Speichern"-Button in der Editieransicht einer Veranstaltung im Backend?
☐ Wenn ja, werden die geänderten Daten gespeichert und wieder die gleiche Editieransicht angezeigt?
- ☐ Gibt es einen "Speichern & Schließen"-Button in der Editieransicht einer Veranstaltung im Backend?
☐ Wenn ja, werden die geänderten Daten gespeichert und die zugehörige Listenansicht angezeigt?
- ☐ Gibt es einen "Speichern & Neu"-Button in der Editieransicht einer Veranstaltung im Backend?
☐ Wenn ja, werden die geänderten Daten gespeichert und eine leere Editieransicht angezeigt?
- ☐ Gibt es einen "Schließen"-Button in der Editieransicht einer Veranstaltung im Backend?
☐ Wenn ja, wird ohne zu speichern die zugehörige Listenansicht angezeigt?

Figure E.21: Requirement A: Test Cases (2)

Ansichten für **Veranstaltungen** im Frontend:

- ☐ Gibt es eine Listenansicht für Veranstaltungen im Frontend?
 - ☐ Wenn ja, kreuzen Sie die vorhandenen Tabellenüberschriften an:
 - ☐ Bezeichnung (dt.) ☐ Verantwortlicher Professor
 - ☐ Können die Listeneinträge über die Spaltenüberschriften sortiert werden?
 - ☐ Gibt es Such- und Filteroptionen für die Listeneinträge (nach Bezeichnung und Professor)?
 - ☐ Kann man in den Menüpunkteinstellungen nach einem bestimmten Studiengang filtern?
- ☐ Gibt es eine Detailansicht für Veranstaltungen im Frontend?
 - ☐ Wenn ja, kreuzen Sie die vorhandenen Felder an:
 - ☐ Bezeichnung (eng.) ☐ Bezeichnung (dt.) ☐ Modulnummer
 - ☐ Typ ☐ Leistungspunkte ☐ Professor ☐ Studiengang ☐ Voraussetzungen
- ☐ Gibt es eine Verlinkung von der Liste aller Veranstaltungen zur Detailansicht einer Veranstaltung im Frontend?

Ansichten für **Fachgebiete**:

- ☐ Gibt es eine Listenansicht für Fachgebiete im Backend?
 - Wenn ja, kreuzen Sie die vorhandenen Tabellenüberschriften an:
 - ☐ ID ☐ Bezeichnung
 - ☐ Können die Listeneinträge über die Spaltenüberschriften sortiert werden?
 - ☐ Gibt es Such- und Filteroptionen für die Listeneinträge?
- ☐ Gibt es eine Editieransicht für Fachgebiete im Backend?
 - ☐ Wenn ja, kreuzen Sie die vorhandenen Felder an:
 - ☐ Bezeichnung
- ☐ Gibt es eine Verlinkung von der Liste aller Fachgebiete zur Editieransicht eines Fachgebiets?

CRUD-Funktionalität für **Fachgebiete**:

- ☐ Gibt es Buttons zum Erstellen, Editieren und Löschen in der Listenansicht von Fachgebieten im Backend?
 - Wenn ja,
 - ☐ Erstellen: Wird zu einer leeren Editieransicht zum Anlegen eines Fachgebiets verlinkt?
 - ☐ Editieren: Wird zu einer Editieransicht zum Editieren eines vorhandenen Fachgebiets verlinkt?
 - ☐ Editieren: Sind die Werte des ausgewählten Fachgebiets bereits in editierbaren Feldern dargestellt?
 - ☐ Löschen: Werden einzeln ausgewählte Einträge erfolgreich gelöscht?
 - ☐ Löschen: Können mehrere Einträge auf einmal gelöscht werden?
- ☐ Gibt es einen "Speichern"-Button in der Editieransicht eines Fachgebiets im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und wieder die gleiche Editieransicht angezeigt?
- ☐ Gibt es einen "Speichern & Schließen"-Button in der Editieransicht eines Fachgebiets im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und die zugehörige Listenansicht angezeigt?
- ☐ Gibt es einen "Speichern & Neu"-Button in der Editieransicht eines Fachgebiets im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und eine leere Editieransicht angezeigt?
- ☐ Gibt es einen "Schließen"-Button in der Editieransicht eines Fachgebiets im Backend?
 - ☐ Wenn ja, wird ohne zu speichern die zugehörige Listenansicht angezeigt?

Ansichten für **Professoren**:

- ☐ Gibt es eine Listenansicht für Professoren im Backend?
 - Wenn ja, kreuzen Sie die vorhandenen Tabellenüberschriften an:
 - ☐ ID ☐ Name ☐ Adresse ☐ Geburtsdatum ☐ Fachbereich ☐ Fachgebiet
 - ☐ Gibt es eine Verlinkung zur Editieransicht der jeweiligen Fachbereiche in der Listenansicht?
 - ☐ Gibt es eine Verlinkung zur Editieransicht der jeweiligen Fachgebiete in der Listenansicht?
 - ☐ Können die Listeneinträge über die Spaltenüberschriften sortiert werden?
 - ☐ Gibt es Such- und Filteroptionen für die Listeneinträge?
- ☐ Gibt es eine Editieransicht für Professoren im Backend?
 - Wenn ja, kreuzen Sie die vorhandenen Felder an:
 - ☐ Name ☐ Adresse ☐ Geburtsdatum
 - ☐ Gibt es ein Feld "Fachbereich" o.ä.?
 - ☐ Wenn ja, kann er aus einer Liste vorhandener Fachbereiche gewählt werden?
 - ☐ Gibt es ein Feld "Fachgebiet" o.ä.?
 - ☐ Wenn ja, kann aus einer Liste vorhandener Fachgebiete ausgewählt werden?
- ☐ Gibt es eine Verlinkung von der Liste aller Professoren zur Editieransicht eines Professors?

CRUD-Funktionalität für **Professoren**:

- ☐ Gibt es Buttons zum Erstellen, Editieren und Löschen in der Listenansicht von Professoren im Backend?
 - Wenn ja,
 - ☐ Erstellen: Wird zu einer leeren Editieransicht zum Anlegen eines Professors verlinkt?
 - ☐ Editieren: Wird zu einer Editieransicht zum Editieren einer vorhandenen Veranstaltung verlinkt?
 - ☐ Editieren: Sind die Werte des ausgewählten Professors bereits in editierbaren Feldern dargestellt?
 - ☐ Löschen: Werden einzeln ausgewählte Einträge erfolgreich gelöscht?
 - ☐ Löschen: Können mehrere Einträge auf einmal gelöscht werden?
- ☐ Gibt es einen "Speichern"-Button in der Editieransicht eines Professors im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und wieder die gleiche Editieransicht angezeigt?
- ☐ Gibt es einen "Speichern & Schließen"-Button in der Editieransicht eines Professors im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und die zugehörige Listenansicht angezeigt?
- ☐ Gibt es einen "Speichern & Neu"-Button in der Editieransicht eines Professors im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und eine leere Editieransicht angezeigt?
- ☐ Gibt es einen "Schließen"-Button in der Editieransicht eines Professors im Backend?
 - ☐ Wenn ja, wird ohne zu speichern die zugehörige Listenansicht angezeigt?

Figure E.22: Requirement A: Test Cases (3)

Anforderung B

Erstellen Sie eine **Joomla-Komponente**, welche ein System zum Customer-Relationship-Management (CRM) für B2B-Kunden realisiert. Berücksichtigen Sie dabei die folgenden **CRUD-Akzeptanzkriterien (Definition of Done)**:

- *Für alle Entitäten gilt, dass je eine Listenansicht sowie eine Editieransicht im Backend verfügbar sein muss.*
- *Die Listenansicht sollen durch Suchoptionen (Suche/Filter) anpassbar sein. Das heißt, die Suche, sowie Filter für alle Felder müssen implementiert werden.*
- *Außerdem müssen die Listeneinträge über die Spaltenüberschriften sortierbar sein. Um lange Listen zu vermeiden, muss die Anzahl der darzustellenden Elemente auswählbar sein (Pagination). Dies gilt für Listenansichten im Backend und Frontend.*
- *Die Ansichten im Backend müssen für alle Entitäten CRUD vollständig anbieten. Im Frontend werden Detailansichten grundsätzlich nicht zum Editieren verwendet - können aber (z.B. durch einen Editier-Button).*
- *Detailansichten und Editieransichten können nur aus dem Kontext von Listenansichten geöffnet werden (z.B. neu oder bearbeiten von vorhandenem Listeneintrag).*

Alle Informationen des CRM-Systems werden über das Backend einer Joomla-Instanz von Mitarbeitern (Admins) verwaltet. Kunden sehen die angebotenen Produkte innerhalb eines Produktkataloges nur im Frontend der Joomla-Seite und können diese über einen Bestellvorgang beziehen. Mitarbeiter pflegen im Backend die Daten für Produkte, Produktkategorien, Kunden, Ansprechpartner, mögliche Zahlungsarten und Bestellungen.

In den Daten eines Kunden sind der Unternehmensname, das Bankkonto, die zugelassenen Zahlungsarten sowie die Liefer- und Rechnungsadresse hinterlegt. Ein Bankkonto enthält die Informationen, welche für ein SEPA-Lastschriftmandat nötig sind. Dazu gehören der Name und die BIC des Kreditinstituts, sowie die IBAN des Kunden. Die Liefer-/Rechnungsadresse enthält Informationen zum Ort, der Postleitzahl, der Straße und dem Land.

Jeder Kunde hat einen oder mehrere Ansprechpartner. Neben den Stammdaten eines Ansprechpartners, wie Vorname, Nachname, Telefonnummer, Email-Adresse kann der CRM-Mitarbeiter den Ansprechpartner einem Kunden zuweisen.

Im Frontend des Shops sehen Kunden den Titel, Preis, die verfügbare Stückzahl und Produktkategorie, sowie die Beschreibung eines Produktes.

Eine Listenansicht im Backend gibt den Mitarbeitern des Shops eine Übersicht über alle Bestellungen. Über eine Schaltfläche "Neu" kann ein neuer Bestellvorgang geöffnet werden. Ein Bestellvorgang enthält eine Liste über alle verfügbaren Produkte. Aus dieser wählt der Mitarbeiter das gewünschte Produkt aus. Anschließend gibt er die zu bestellende Menge an und wählt den Kunden aus. Nachdem alle Eingaben getätigt wurden kann der Mitarbeiter den Bestellvorgang über die Speichern-Schaltfläche abschließen. Über eine entsprechende Schaltfläche kann der Mitarbeiter den Bestellvorgang ggf. auch abbrechen.

Ein Mitarbeiter kann eine Bestellung anklicken und sieht alle Informationen zu der ausgewählten Bestellung. Er kann außerdem den Status der Bestellung von "Offen" in "In Bearbeitung" und "Geschlossen" ändern.

Erstellen Sie neben der Komponente ein **Joomla-Modul**, welches den Status aller Bestellungen anzeigt. In dem Modul sollen nur die Auftragsnummer und der Status jeder Bestellung angezeigt werden.

Figure E.23: Requirement B: Customer-Relationship Management

Akzeptanzkriterien - Checkliste:

Struktur der Erweiterungen:

- ☐ Ist die Komponente installierbar?
- ☐ Ist ein Update-Skript implementiert?
- ☐ Ist die Mehrsprachigkeit (durch Sprachkonstanten und -dateien) realisiert?

- ☐ Ist das Modul implementiert?
- ☐ Ist das Modul installierbar?
- ☐ Stellt das Modul die gewünschten Daten der Komponente dar?

Ansichten für Produktkategorien:

- ☐ Gibt es eine Listenansicht für Produktkategorien im Backend?
Wenn ja, kreuzen Sie die vorhandenen Tabellenüberschriften an:
 - ☐ ID ☐ Bezeichnung
 - ☐ Können die Listeneinträge über die Spaltenüberschriften sortiert werden?
 - ☐ Gibt es Such- und Filteroptionen für die Listeneinträge?
- ☐ Gibt es eine Editieransicht für Produktkategorien im Backend?
Wenn ja, kreuzen Sie die vorhandenen Felder an:
 - ☐ Bezeichnung
- ☐ Gibt es eine Verlinkung von der Liste aller Produktkategorien zur Editieransicht einer Produktkategorie im Backend?

CRUD-Funktionalität für Produktkategorien:

- ☐ Gibt es Buttons zum Erstellen, Editieren und Löschen in der Listenansicht von Produktkategorien im Backend?
Wenn ja,
 - ☐ Erstellen: Wird zu einer leeren Editieransicht zum Anlegen einer Produktkategorie verlinkt?
 - ☐ Editieren: Wird zu einer Editieransicht zum Editieren einer vorhandenen Produktkategorie verlinkt?
 - ☐ Editieren: Sind die Werte der ausgewählten Produktkategorie bereits in editierbaren Feldern dargestellt?
 - ☐ Löschen: Werden einzeln ausgewählte Einträge erfolgreich gelöscht?
 - ☐ Löschen: Können mehrere Einträge auf einmal gelöscht werden?
- ☐ Gibt es einen "Speichern"-Button in der Editieransicht einer Produktkategorie im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und wieder die gleiche Editieransicht angezeigt?
- ☐ Gibt es einen "Speichern & Schließen"-Button in der Editieransicht einer Produktkategorie im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und die zugehörige Listenansicht angezeigt?
- ☐ Gibt es einen "Speichern & Neu"-Button in der Editieransicht einer Produktkategorie im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und eine leere Editieransicht angezeigt?
- ☐ Gibt es einen "Schließen"-Button in der Editieransicht einer Produktkategorie im Backend?
 - ☐ Wenn ja, wird ohne zu speichern die zugehörige Listenansicht angezeigt?

Ansichten für Produkte im Backend:

- ☐ Gibt es eine Listenansicht für Produkte im Backend?
Wenn ja, kreuzen Sie die vorhandenen Tabellenüberschriften an:
 - ☐ ID ☐ Bezeichnung ☐ Beschreibung ☐ Preis ☐ Stückzahl ☐ Produktkategorie
 - ☐ Gibt es eine Verlinkung zur Editieransicht der jeweiligen Produktkategorien in der Listenansicht?
 - ☐ Können die Listeneinträge über die Spaltenüberschriften sortiert werden?
 - ☐ Gibt es Such- und Filteroptionen für die Listeneinträge?
- ☐ Gibt es eine Editieransicht für Produkte im Backend?
Wenn ja, kreuzen Sie die vorhandenen Felder an:
 - ☐ Bezeichnung ☐ Beschreibung ☐ Preis ☐ Stückzahl
 - ☐ Gibt es ein Feld "Produktkategorie" o.ä.?
 - ☐ Wenn ja, kann aus einer Liste vorhandener Produktkategorien gewählt werden?
- ☐ Gibt es eine Verlinkung von der Liste aller Produkte zur Editieransicht eines Produkts im Backend?

CRUD-Funktionalität für Produkte im Backend:

- ☐ Gibt es Buttons zum Erstellen, Editieren und Löschen in der Listenansicht von Produkten im Backend?
Wenn ja,
 - ☐ Erstellen: Wird zu einer leeren Editieransicht zum Anlegen eines Produkts verlinkt?
 - ☐ Editieren: Wird zu einer Editieransicht zum Editieren eines vorhandenen Produkts verlinkt?
 - ☐ Editieren: Sind die Werte des ausgewählten Produkts bereits in editierbaren Feldern dargestellt?
 - ☐ Löschen: Werden einzeln ausgewählte Einträge erfolgreich gelöscht?
 - ☐ Löschen: Können mehrere Einträge auf einmal gelöscht werden?
- ☐ Gibt es einen "Speichern"-Button in der Editieransicht eines Produkts im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und wieder die gleiche Editieransicht angezeigt?
- ☐ Gibt es einen "Speichern & Schließen"-Button in der Editieransicht eines Produkts im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und die zugehörige Listenansicht angezeigt?
- ☐ Gibt es einen "Speichern & Neu"-Button in der Editieransicht eines Produkts im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und eine leere Editieransicht angezeigt?
- ☐ Gibt es einen "Schließen"-Button in der Editieransicht eines Produkts im Backend?
 - ☐ Wenn ja, wird ohne zu speichern die zugehörige Listenansicht angezeigt?

Figure E.24: Requirement B: Test Cases (1)

Ansichten für Produktkataloge im Frontend:

- ☐ Gibt es eine Listenansicht für Produkte im Frontend?
 - ☐ Wenn ja, kreuzen Sie die vorhandenen Tabellenüberschriften an:
 - ☐ Bezeichnung ☐ Produktkategorie ☐ Preis ☐ Stückzahl
 - ☐ Können die Listeneinträge über die Spaltenüberschriften sortiert werden?
 - ☐ Gibt es Such- und Filteroptionen für die Listeneinträge?
- ☐ Gibt es eine Detailansicht für Produkte im Frontend?
 - ☐ Wenn ja, kreuzen Sie die vorhandenen Felder an:
 - ☐ Bezeichnung ☐ Beschreibung ☐ Preis ☐ Stückzahl ☐ Produktkategorie
- ☐ Gibt es eine Verlinkung von der Liste aller Produkte zur Detailansicht eines Produkts im Frontend?

Ansichten für Zahlungsarten:

- ☐ Gibt es eine Listenansicht für Zahlungsarten im Backend?
 - Wenn ja, kreuzen Sie die vorhandenen Tabellenüberschriften an:
 - ☐ ID ☐ Bezeichnung
 - ☐ Können die Listeneinträge über die Spaltenüberschriften sortiert werden?
 - ☐ Gibt es Such- und Filteroptionen für die Listeneinträge?
- ☐ Gibt es eine Editieransicht für Zahlungsarten im Backend?
 - Wenn ja, kreuzen Sie die vorhandenen Felder an:
 - ☐ Bezeichnung
 - ☐ Gibt es ein Feld "Gewährt" o.ä.?
 - ☐ Wenn ja, können beliebig viele Elemente aus einer Liste vorhandener Kunden gewählt werden?
- ☐ Gibt es eine Verlinkung von der Liste aller Zahlungsarten zur Editieransicht einer Zahlungsarten im Backend?

CRUD-Funktionalität für Zahlungsarten:

- ☐ Gibt es Buttons zum Erstellen, Editieren und Löschen in der Listenansicht von Zahlungsarten im Backend?
 - Wenn ja,
 - ☐ Erstellen: Wird zu einer leeren Editieransicht zum Anlegen einer Zahlungsart verlinkt?
 - ☐ Editieren: Wird zu einer Editieransicht zum Editieren einer vorhandenen Zahlungsart verlinkt?
 - ☐ Editieren: Sind die Werte der ausgewählten Zahlungsart bereits in editierbaren Feldern dargestellt?
 - ☐ Löschen: Werden einzeln ausgewählte Einträge erfolgreich gelöscht?
 - ☐ Löschen: Können mehrere Einträge auf einmal gelöscht werden?
- ☐ Gibt es einen "Speichern"-Button in der Editieransicht einer Zahlungsart im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und wieder die gleiche Editieransicht angezeigt?
- ☐ Gibt es einen "Speichern & Schließen"-Button in der Editieransicht einer Zahlungsart im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und die zugehörige Listenansicht angezeigt?
- ☐ Gibt es einen "Speichern & Neu"-Button in der Editieransicht einer Zahlungsart im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und eine leere Editieransicht angezeigt?
- ☐ Gibt es einen "Schließen"-Button in der Editieransicht einer Zahlungsart im Backend?
 - ☐ Wenn ja, wird ohne zu speichern die zugehörige Listenansicht angezeigt?

Ansichten für Kunden:

- ☐ Gibt es eine Listenansicht für Kunden im Backend?
 - Wenn ja, kreuzen Sie die vorhandenen Tabellenüberschriften an:
 - ☐ ID ☐ Unternehmensname ☐ Lieferadresse ☐ Rechnungsadresse
 - ☐ Bankname ☐ BIC ☐ IBAN ☐ Zahlungsarten
 - ☐ Werden alle zugewiesenen Zahlungsarten für jeden Kunden in der Listenansicht angezeigt?
 - ☐ Wenn ja, gibt es eine Verlinkung zu Zahlungsarten in der Listenansicht?
 - ☐ Können die Listeneinträge über die Spaltenüberschriften sortiert werden?
 - ☐ Gibt es Such- und Filteroptionen für die Listeneinträge?
- ☐ Gibt es eine Editieransicht für Kunden im Backend?
 - Wenn ja, kreuzen Sie die vorhandenen Felder an:
 - ☐ Unternehmensname ☐ Lieferadresse ☐ Rechnungsadresse
 - ☐ Bankname ☐ BIC ☐ IBAN
 - ☐ Gibt es ein Feld "Zahlungsarten" o.ä.?
 - ☐ Wenn ja, können beliebig viele Elemente aus einer Liste vorhandener Zahlungsarten gewählt werden?
- ☐ Gibt es eine Verlinkung von der Liste aller Kunden zur Editieransicht eines Kunden im Backend?

CRUD-Funktionalität für Kunden:

- ☐ Gibt es Buttons zum Erstellen, Editieren und Löschen in der Listenansicht von Kunden im Backend?
 - Wenn ja,
 - ☐ Erstellen: Wird zu einer leeren Editieransicht zum Anlegen eines Kunden verlinkt?
 - ☐ Editieren: Wird zu einer Editieransicht zum Editieren eines vorhandenen Kunden verlinkt?
 - ☐ Editieren: Sind die Werte des ausgewählten Kunden bereits in editierbaren Feldern dargestellt?
 - ☐ Löschen: Werden einzeln ausgewählte Einträge erfolgreich gelöscht?
 - ☐ Löschen: Können mehrere Einträge auf einmal gelöscht werden?
- ☐ Gibt es einen "Speichern"-Button in der Editieransicht eines Kunden im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und wieder die gleiche Editieransicht angezeigt?
- ☐ Gibt es einen "Speichern & Schließen"-Button in der Editieransicht eines Kunden im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und die zugehörige Listenansicht angezeigt?
- ☐ Gibt es einen "Speichern & Neu"-Button in der Editieransicht eines Kunden im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und eine leere Editieransicht angezeigt?
- ☐ Gibt es einen "Schließen"-Button in der Editieransicht eines Kunden im Backend?
 - ☐ Wenn ja, wird ohne zu speichern die zugehörige Listenansicht angezeigt?

Figure E.25: Requirement B: Test Cases (2)

Ansichten für **Ansprechpartner**:

- ☐ Gibt es eine Listenansicht für Ansprechpartner im Backend?
Wenn ja, kreuzen Sie die vorhandenen Tabellenüberschriften an:
☐ ID ☐ Vorname ☐ Nachname ☐ Telefonnummer ☐ E-Mail ☐ Unternehmen
 - ☐ Wenn ja, gibt es eine Verlinkung zur Editieransicht der jeweiligen Kunden (Unternehmen) in der Listenansicht?
 - ☐ Können die Listeneinträge über die Spaltenüberschriften sortiert werden?
 - ☐ Gibt es Such- und Filteroptionen für die Listeneinträge?
- ☐ Gibt es eine Editieransicht für Ansprechpartner im Backend?
Wenn ja, kreuzen Sie die vorhandenen Felder an:
☐ Vorname ☐ Nachname ☐ Telefonnummer ☐ E-Mail
 - ☐ Gibt es ein Feld "Unternehmen" o.ä.?
 - ☐ Wenn ja, kann aus einer Liste vorhandener Kunden (Unternehmen) gewählt werden?
- ☐ Gibt es eine Verlinkung von der Liste aller Ansprechpartner zur Editieransicht eines Ansprechpartners im Backend?

CRUD-Funktionalität für **Ansprechpartner**:

- ☐ Gibt es Buttons zum Erstellen, Editieren und Löschen in der Listenansicht von Ansprechpartnern im Backend?
Wenn ja,
 - ☐ Erstellen: Wird zu einer leeren Editieransicht zum Anlegen eines Ansprechpartners verlinkt?
 - ☐ Editieren: Wird zu einer Editieransicht zum Editieren eines vorhandenen Ansprechpartners verlinkt?
 - ☐ Editieren: Sind die Werte des ausgewählten Ansprechpartners bereits in editierbaren Feldern dargestellt?
 - ☐ Löschen: Werden einzeln ausgewählte Einträge erfolgreich gelöscht?
 - ☐ Löschen: Können mehrere Einträge auf einmal gelöscht werden?
- ☐ Gibt es einen "Speichern"-Button in der Editieransicht eines Ansprechpartners im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und wieder die gleiche Editieransicht angezeigt?
- ☐ Gibt es einen "Speichern & Schließen"-Button in der Editieransicht eines Ansprechpartners im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und die zugehörige Listenansicht angezeigt?
- ☐ Gibt es einen "Speichern & Neu"-Button in der Editieransicht eines Ansprechpartners im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und eine leere Editieransicht angezeigt?
- ☐ Gibt es einen "Schließen"-Button in der Editieransicht eines Ansprechpartners im Backend?
 - ☐ Wenn ja, wird ohne zu speichern die zugehörige Listenansicht angezeigt?

Ansichten für **Bestellungen**:

- ☐ Gibt es eine Listenansicht für Bestellungen im Backend?
Wenn ja, kreuzen Sie die vorhandenen Tabellenüberschriften an:
☐ ID ☐ Status ☐ Produkt ☐ Menge ☐ Kunde
 - ☐ Wenn ja, gibt es eine Verlinkung zur Editieransicht der jeweiligen Produkte in der Listenansicht?
 - ☐ Wenn ja, gibt es eine Verlinkung zur Editieransicht der jeweiligen Kunden in der Listenansicht?
 - ☐ Können die Listeneinträge über die Spaltenüberschriften sortiert werden?
 - ☐ Gibt es Such- und Filteroptionen für die Listeneinträge?
- ☐ Gibt es eine Editieransicht für Bestellungen im Backend?
Wenn ja, kreuzen Sie die vorhandenen Felder an:
☐ Status ☐ Menge
 - ☐ Gibt es ein Feld "Produkt" o.ä.?
 - ☐ Wenn ja, kann aus einer Liste vorhandener Produkte gewählt werden?
 - ☐ Gibt es ein Feld "Kunde" o.ä.?
 - ☐ Wenn ja, kann aus einer Liste vorhandener Kunden gewählt werden?
- ☐ Gibt es eine Verlinkung von der Liste aller Bestellungen zur Editieransicht einer Bestellung im Backend?

CRUD-Funktionalität für **Bestellung**:

- ☐ Gibt es Buttons zum Erstellen, Editieren und Löschen in der Listenansicht von Bestellungen im Backend?
Wenn ja,
 - ☐ Erstellen: Wird zu einer leeren Editieransicht zum Anlegen einer Bestellung verlinkt?
 - ☐ Editieren: Wird zu einer Editieransicht zum Editieren einer vorhandenen Bestellung verlinkt?
 - ☐ Editieren: Sind die Werte der ausgewählten Bestellung bereits in editierbaren Feldern dargestellt?
 - ☐ Löschen: Werden einzeln ausgewählte Einträge erfolgreich gelöscht?
 - ☐ Löschen: Können mehrere Einträge auf einmal gelöscht werden?
- ☐ Gibt es einen "Speichern"-Button in der Editieransicht einer Bestellung im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und wieder die gleiche Editieransicht angezeigt?
- ☐ Gibt es einen "Speichern & Schließen"-Button in der Editieransicht einer Bestellung im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und die zugehörige Listenansicht angezeigt?
- ☐ Gibt es einen "Speichern & Neu"-Button in der Editieransicht einer Bestellung im Backend?
 - ☐ Wenn ja, werden die geänderten Daten gespeichert und eine leere Editieransicht angezeigt?
- ☐ Gibt es einen "Schließen"-Button in der Editieransicht einer Bestellung im Backend?
 - ☐ Wenn ja, wird ohne zu speichern die zugehörige Listenansicht angezeigt?

Figure E.26: Requirement B: Test Cases (3)

F

Controlled Experiment: Results

In this appendix, the detailed results of the controlled experiment are collected (cf. Section 7.2). To anonymize the results, the subjects got a random subject-ID, which they had to put on each document. We use this subject-ID in each table below to illustrate the row data for each subject.

Subjects

At the beginning of the experiment, the subjects received questionnaires in order to get insight about their *developer experience* (see Table F.1 and Table F.2), their *experience with a WCMS and Joomla* (see Table F.3), and their *experience with MDE* (see Table F.4). Moreover, we asked questions in order to get insight about the *open-mindedness towards MDE* (see Table F.5).

Table F.1: Developer Experience (1)

Participant	Self-perception (inexperienced - very experienced)	Experience as Software Developer (years)	Experience with Web Technologies
xmA2FN	experienced	2-5	high
E7cPAZ	experienced	2-5	satisfactory
medNCR	very experienced	6-10	high
S4aA8A	experienced	2-5	satisfactory
79nd5v	less experienced	2-5	satisfactory
iEnQwP	experienced	2-5	high
QWsYAL	experienced	6-10	high
s4H6Gg	experienced	6-10	high
CN4w9N	experienced	2-5	high
745tzX	experienced	2-5	high
g8S6vb	experienced	2-5	high
jr6rC9	experienced	11-20	high
yR2tS3	experienced	6-10	satisfactory
HwmYff	experienced	11-20	satisfactory

Table F.2: Developer Experience (2)

Participant	Used Development Environments	Developed Software based on given Requirements
xmA2FN	JetBrains	yes
E7cPAZ	JetBrains, VisualStudio Code	yes
medNCR	JetBrains	no
S4aA8A	JetBrains, VisualStudio Code, Xcode	yes
79nd5v	JetBrains	yes
iEnQwP	JetBrains, VisualStudio Code	yes
QWsYAL	JetBrains	yes
s4H6Gg	JetBrains	yes
CN4w9N	Eclipse, JetBrains	yes
745tzX	JetBrains, Brackets	yes
g8S6vb	JetBrains, VisualStudio Code, VIM	yes
jr6rC9	Eclipse, JetBrains	yes
yR2tS3	VIM, Notepad++, VisualStudio Code	yes
HwmYff	Eclipse, JetBrains	yes

Table F.3: Experience with a WCMS and Joomla

Participant	CMS Experience			Joomla Experience		
	<i>Years</i>	<i>Role (User, Administrator, Developer)</i>	<i>Years</i>	<i>Scenario 2</i>	<i>External Assessment Score</i>	
xmA2FN	2-5	U, A, D	<1	yes	89%	
E7cPAZ	2-5	D	<1	no	100%	
medNCR	6-10	U, A, D	6-10	yes	67%	
S4aA8A	2-5	U, A	<1	no	68%	
79nd5v	2-5	D	<1	yes	89%	
iEnQwP	2-5	U, A, D	2-5	no	85%	
QWsYAL	<1	D	<1	no	53%	
s4H6Gg	2-5	D	<1	no	56%	
CN4w9N	2-5	U, A, D	<1	no	41%	
745tzX	6-10	U, A, D	<1	no	33%	
g8S6vb	<1	U, A, D	<1	no	93%	
jr6rC9	6-10	A, D	6-10	yes	81%	
yR2tS3	2-5	D	2-5	no	50%	
HwmYff	6-10	U, A, D	2-5	no	64%	

Table F.4: Experience with MDE

Participant	Experience with Modelling (years)	Languages	Self-perception (none - very high)	External Assessment Score
xmA2FN	2-5	n.a.	satisfactory	93%
E7cPAZ	2-5	UML	sufficient	80%
medNCR	<1	UML	good	88%
S4aA8A	2-5	UML	satisfactory	80%
79nd5v	n.a.	UML	poor	73%
iEnQwP	<1	UML	sufficient	83%
QWsYAL	2-5	UML, EMF Ecore, MPS	sufficient	93%
s4H6Gg	2-5	UML	sufficient	80%
CN4w9N	2-5	UML	sufficient	90%
745tzX	2-5	UML	satisfactory	75%
g8S6vb	<1	UML	poor	80%
jr6rC9	2-5	UML, EMF Ecore	satisfactory	85%
yR2tS3	2-5	UML, eJSL (JooMDD)	sufficient	75%
HwmYfF	2-5	UML, eJSL (JooMDD)	sufficient	85%

Table F.5: Open-Mindedness towards MDE

Participant	Estimation of MDE Relevance	Experience with Code Generation
xmA2FN	very high	none
E7cPAZ	very high	UML → code (classes/methods/interfaces)
medNCR	high	none
S4aA8A	average	Generation of Documentation
79nd5v	high	none
iEnQwP	high	Generation of Documentation
QWsYAL	average	Generation of Documentation
s4H6Gg	high	Generation of Documentation
CN4w9N	average	none
745tzX	high	Generation of Documentation
g8S6vb	very high	Generation of Joomla Extensions
jr6rC9	high	Generation of Joomla Extensions
yR2tS3	high	Generation of Joomla Extensions
HwmYfF	average	Generation of Documentation/Joomla Extensions/Mobile Applications

Development Session 1

In Table F.6 we present the row data of development session 1 (traditional development). This includes the measurement results for productivity based on the ratio of passed test cases (accuracy). Both requirements (A and B) hat 28 test cases for 4 requirement groups in total (cf. the requirement checklists in Appendix E).

Table F.6: Session 1: Productivity Results (Row Data)

Participant	Requirement	Structure Component	Views	CRUD	Structure Module	Overall
xmA2FN	B	25%	0%	0%	0%	0.9%
E7cPAZ	A	75%	0%	0%	0%	2.7%
medNCR	B	75%	0%	0%	0%	2.7%
S4aA8A	A	75%	14.3%	16.7%	0%	17%
79nd5v	B	25%	0%	0%	0%	0.9%
iEnQwP	A	50%	5.4%	4.2%	0%	6.3%
QWsYAL	A	25%	5.9%	12.5%	0%	9.2%
s4H6Gg	B	75%	12.5%	16.7%	0%	16.1%
CN4w9N	A	75%	14.3%	16.7%	0%	17%
745tzX	B	75%	14.3%	12.5%	0%	15.2%
g8S6vb	B	100%	0%	0%	0%	3.6%
jr6rC9	A	50%	0%	0%	0%	1.8%
yR2tS3	B	100%	12.5%	41.7%	0%	27.7%
HwmYfF	A	25%	0%	0%	0%	0.9%

Table F.7 below shows the row data of the quality measurement results of development session 1 (traditional development). For each submitted solution we counted the amount of views, corresponding files and LoC, and violations in order to calculate the *violations to LoC ratio*. One outlier exists (highlighted row). This is traced to the fact, that the subject used a code generator during the first session.

Table F.7: Session 1: Quality Results (Row Data)

Participant	Requirement	Views	Files	LoC	Violations (PSR-2)	$\frac{\text{Violations}}{\text{LoC}}$
xmA2FN	B	2	4	113	5	4.42%
E7cPAZ	A	2	7	461	8	1.74%
medNCR	B	6	16	1794	14	0.78%
S4aA8A	A	2	9	1430	8	0.56
79nd5v	B	0	0	0	0	0%
iEnQwP	A	2	10	563	9	1.6%
QWsYAL	A	2	8	450	7	1.56%
s4H6Gg	B	2	9	570	9	1.58%
CN4w9N	A	2	11	551	9	1.63%
745tzX	B	2	11	542	9	1.66%
g8S6vb	B	0	0	0	0	0%
jr6rC9	A	0	0	0	0	0%
yR2tS3	B	12	104	16763	68	0.41%
HwmYfF	A	0	0	0	0	0%

After the session, we asked the subjects, if they implemented an extension with similar complexity before and which development environment they used (see Table F.8). Additionally, the subjects had to describe their development approach (see Table F.9). We asked, if they developed the extension from scratch, applied a clone-and-own approach, and oriented on existing extensions. Moreover, we asked, if the subjects made use of an existing code generator.

Table F.8: Session 1: Session Feedback (Row Data)

Participant	Implemented an Extension with similar Complexity before	Development Environment
xmA2FN	no	none
E7cPAZ	yes	JetBrains
medNCR	yes	JetBrains
S4aA8A	yes	JetBrains
79nd5v	no	JetBrains
iEnQwP	no	JetBrains
QWsYAL	yes	JetBrains
s4H6Gg	yes	JetBrains
CN4w9N	yes	JetBrains
745tzX	yes	JetBrains
g8S6vb	yes	JetBrains, VisualStudio Code
jr6rC9	yes	JetBrains
yR2tS3	no, not for Joomla	none
HwmYfF	yes	JetBrains

Table F.9: Session 1: Session Feedback - Development Approach (Row Data)

Participant	From Scratch	Clone-and-Own	Oriented on Existing Extension	Used Code Generator
xmA2FN	yes	yes	3rd-party	component-creator.com
E7cPAZ	yes	no	no	no
medNCR	no	yes	yes	no
S4aA8A	yes	yes	no	component-creator
79nd5v	yes	no	3rd-party, core	no
iEnQwP	yes	yes	core	no
QWsYAL	yes	no	no	no
s4H6Gg	no	no	own	no
CN4w9N	no	yes	no	no
745tzX	yes	yes	own, 3rd-party, core	no
g8S6vb	no	yes	no	no
jr6rC9	yes	no	no	no
yR2tS3	no	no	no	Joomla Component Builder
HwmYfF	yes	no	no	no

Development Session 2

Table F.10 summarizes the row data of development session 2 (MDE). Again, this includes the measurement results for productivity based on the ratio of passed test cases (accuracy). Two measurement results can be interpreted as outliers (highlighted rows).

Table F.10: Session 2: Productivity Results (Row Data)

Participant	Requirement	Structure Component	Views	CRUD	Structure Module	Overall
xmA2FN	A	100%	74.5%	100%	0%	83.7%
E7cPAZ	B	100%	55.2%	66.7%	0%	59.7%
medNCR	A	100%	90.2%	100%	100%	95.1%
S4aA8A	B	100%	61.3%	58.3%	0%	59.2%
79nd5v	A	100%	89%	100%	0%	90.9%
iEnQwP	B	100%	97.3%	100%	100%	98.7%
QWsYAL	B	0%	13.1%	16.7%	0%	13.7%
s4H6Gg	A	100%	92%	83.3%	100%	88.8%
CN4w9N	B	100%	93.5%	83.3%	100%	89.6%
745tzX	A	100%	91.7%	100%	100%	95.8%
g8S6vb	A	100%	86.3%	100%	0%	89.6%
jr6rC9	B	100%	61.9%	66.7%	0%	63.1%
yR2tS3	A	100%	92.6%	100%	66.7%	95.1%
HwmYfF	B	0%	7.1%	10.4%	0%	8%

Table F.11 includes the row data of the quality measurement results of development session 2 (MDE). Similar to session 1, we counted the amount of views, corresponding files and LoC, and violations in order to calculate the *violations to LoC ratio*. However, during the measurement, we only considered the same views, which were implemented during session 1.

Table F.11: Session 2: Quality Results (Row Data)

Participant	Requirement	Views	Files	LoC	Violations (PSR-2)	$\frac{\text{Violations}}{\text{LoC}}$
xmA2FN	A	4	20	2374	16	0.67%
E7cPAZ	B	10	50	5468	40	0.73%
medNCR	A	4	20	2760	16	0.58%
S4aA8A	B	8	40	4766	32	0.67%
79nd5v	A	4	20	2331	16	0.69%
iEnQwP	B	12	60	7458	48	0.64%
QWsYAL	B	12	60	7795	48	0.62%
s4H6Gg	A	4	20	2323	16	0.69%
CN4w9N	B	12	60	6781	48	0.71%
745tzX	A	4	20	2403	16	0.67%
g8S6vb	A	4	20	2335	16	0.69%
jr6rC9	B	8	40	4768	32	0.67%
yR2tS3	A	4	20	2394	16	0.67%
HwmYfF	B	2	10	1228	15	1.22%

Moreover, we measured the code style violations also for submitted modules after the MDE session (see Table F.12). So, the data can be compared e.g. with results of further instantiations of the experiment.

Table F.12: Session 2: Quality Results - Modules (Row Data)

Participant	Requirement	Files	LoC	Violations (PSR-2)	$\frac{\text{Violations}}{\text{LoC}}$
xmA2FN	A	0	0	0	0%
E7cPAZ	B	0	0	0	0%
medNCR	A	3	261	1	0.38%
S4aA8A	B	0	0	0	0%
79nd5v	A	0	0	0	0%
iEnQwP	B	0	0	0	0%
QWsYAL	B	3	247	1	0.4%
s4H6Gg	A	0	0	0	0%
CN4w9N	B	0	0	0	0%
745tzX	A	3	281	1	0.36%
g8S6vb	A	3	255	1	0.39%
jr6rC9	B	0	0	0	0%
yR2tS3	A	0	0	0	0%
HwmYff	B	0	0	0	0%

After the session, we asked the subjects, if all requirements could be expressed using the given in modelling language (eJSL) and if they had to refine the generated code by hand (see Table F.13).

Table F.13: Session 2: Session Feedback (Row Data)

Participant	All Requirements could be expressed with the DSL	Refined Generated Code
xmA2FN	no	no
E7cPAZ	no	no
medNCR	no	yes
S4aA8A	yes	no
79nd5v	no	no
iEnQwP	no	no
QWsYAL	yes	no
s4H6Gg	yes	no
CN4w9N	yes	no
745tzX	no	no
g8S6vb	yes	no
jr6rC9	no	no
yR2tS3	yes	no
HwmYff	yes	no

Feedback

In Table F.14 and Table F.15 we present the feedback of the subjects which was questioned after both sessions. The questions can be found in Appendix E.

Table F.14: Feedback Results - Experiment (Row Data)


Participant	Enough Time for the Tasks	Experimental Environment affected the Development	Free Text
xmA2FN	too little	no	-
E7cPAZ	too little	no	-
medNCR	appropriate	yes, during both sessions	-
S4aA8A	appropriate	yes, during session 2	<i>Es hat sehr viel Spaß gemacht, mit JooMDD zu arbeiten. Eine wirklich einfache Sprache mit mächtigem Output.</i>
79nd5v	too little	no	<i>Super ;)</i>
iEnQwP	too little	yes, during both sessions	<i>Die Doku ist noch ausbaufähig und die Fehler manchmal zu generisch (manchmal aber super hilfreich!). Der Web-Editor ist praktisch. Modellbeispiele waren hilfreich. Ebenso die Experiment-Umgebung, sonst wäre ich beim 1. Teil depressiv geworden.</i>
QWsYAL	too little	no	<i>JooMDD ist sehr gut!</i>
s4H6Gg	too little	no	<i>Zu wenig Zeit für Teil 1, passend für Teil 2</i>
CN4w9N	appropriate	yes, during session 2	<i>Konventionelle Joomla-Entwicklung hätte 3 Tage mehr beansprucht Einführung in JooMDD mit Beispielen, bei Joomla (konventionell) nicht.</i>
745tzX	appropriate	no	<i>Für native Joomla!-Entwicklung wäre die Zeit etwa 3- 8 Tage zu knapp bemessen</i>
g8S6vb	-	no	<i>Dummy-Daten wären cool (entsprechend der Typen, aktivierbar mit einem Flag oder sowas) Selects wären optional nützlich, warum muss ich das hart über eine Referenz regeln, wenn es auch ohne geht. WebEditor super Sache und Kriegsentscheidend für die Verwendung bzw. das Marketing von JooMDD</i>
jr6rC9	too little	yes, during session 1	<i>:-)</i>
yR2tS3	too little	no	-
HwmYfF	too little	no	-


Table F.15: Feedback Results - MDE (Row Data)

Participant	Covered more Requirements with MDE Infrastructure	Development with MDE Tools was more comfortable
xmA2FN	yes	yes
E7cPAZ	yes	yes
medNCR	yes	yes
S4aA8A	yes	yes
79nd5v	yes	yes
iEnQwP	yes	yes
QWsYAL	yes	yes
s4H6Gg	yes	yes
CN4w9N	yes	yes
745tzX	yes	yes
g8S6vb	yes	yes
jr6rC9	yes	yes
yR2tS3	yes	yes
HwmYfF	yes	yes



In this appendix, we illustrate the presentation of the hands-on tutorial which was held in Dongen, Netherlands on December 12, 2017.


 Institut für
Informationswissenschaften


THM
 TECHNISCHE HOCHSCHULE MITTELHESSEN

JOOMDD

DENNIS PRIEFER, WOLF ROST, PETER KNEISEL

DEVELOPER MEETING, DONGEN, THE NETHERLANDS

1. CHALLENGES
2. MODEL-DRIVEN DEVELOPMENT
3. SESSION

CHALLENGES

- Adhere to the required extension structure
 - Large amount of boilerplate code
 - Hard for new developers
- Changes of underlying technology
 - Migration to new version
 - Migration to new system
- Augmentation of existing extensions
 - Reverse engineering of existing extension
 - Dependencies to legacy code

Component

```

graph TD
    Data[Data] --> Module[Module]
    Module --> Model[Model]
    Model --> View[View]
    View --> Template[Template]
    Template --> Helper[Helper]
    Helper --> Module
    
```

Module

```

graph TD
    Data[Data] --> Module[Module]
    Module --> Model[Model]
    Model --> View[View]
    View --> Template[Template]
    
```

Component A

```

graph TD
    Data[Data] --> Module[Module]
    Module --> Model[Model]
    Model --> View[View]
    View --> Template[Template]
    
```

Component B

```

graph TD
    Data[Data] --> Module[Module]
    Module --> Model[Model]
    Model --> View[View]
    View --> Template[Template]
    
```

DENNIS PRIEFER, WOLF ROST 09.12.17

Figure G.1: Hands-on Tutorial Presentation (1)

MODEL-DRIVEN DEVELOPMENT OF JOOMLA! EXTENSIONS

Domain-Specific Language

Code generator

Web editor with model extractor

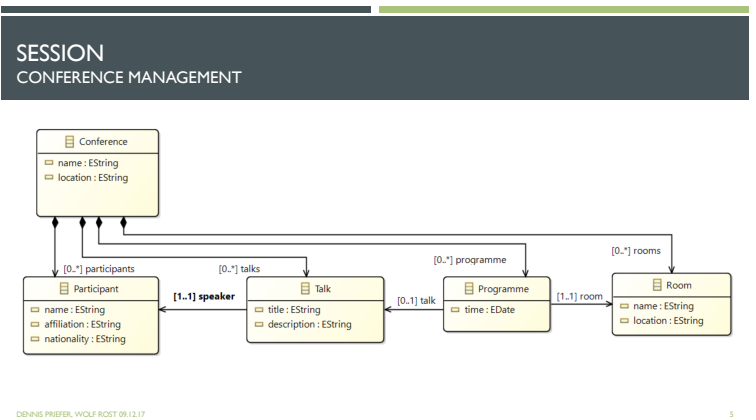
IDE plugins:

- Eclipse
- IntelliJ IDEA
- PhpStorm

J!MDD

SESSION

Copyright © 2000 United Feature Syndicate, Inc.
Redistribution in whole or in part prohibited



SESSION CONFERENCE MANAGEMENT

Figure G.2: Hands-on Tutorial Presentation (2)

SESSION CONFERENCE MANAGEMENT

You are here: [Home](#) > [Talks](#)

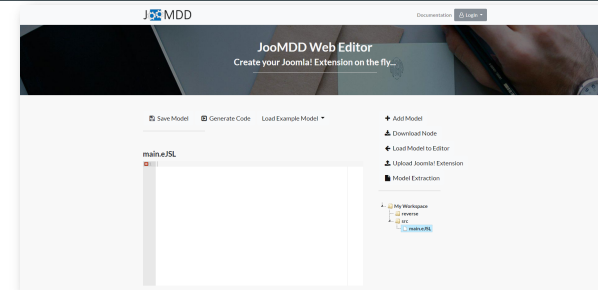
Search

Title	Speaker
JooMDD - Simple Joomla Extension Creation	Dennis Priefer
iCampus Joomla Extensions	Wolf Rost
Design Patterns	John Doe
Internet of Things	Yaddow Green

DENNIS PRIEFER, WOLF ROST 09.12.17

7

SESSION THE EDITOR



DENNIS PRIEFER, WOLF ROST 09.12.17

8

SESSION THE EDITOR

Documentation

Username

Password

New around here? Sign up

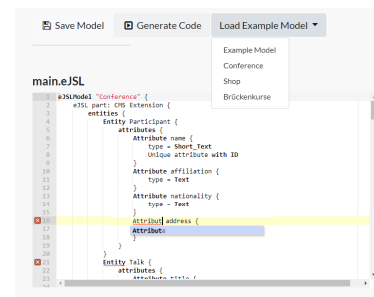
odel

- Test the editor without an account
- Login to save your models and generated code

DENNIS PRIEFER, WOLF ROST 09.12.17

9

SESSION THE EDITOR



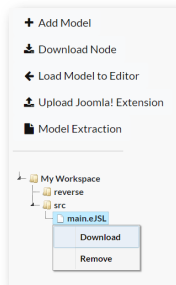
- Example models
- Auto completion
- Syntax highlighting
- Live validation
- Trigger code generation

DENNIS PRIEFER, WOLF ROST 09.12.17

10

Figure G.3: Hands-on Tutorial Presentation (3)

SESSION THE EDITOR

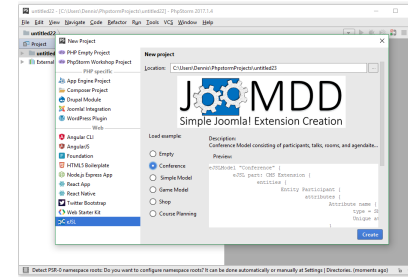


DENNIS PRIEFER, WOLFGANG ROST 09.12.17

- File Tree
- Download of sub nodes via button and context menu
- Upload of existing Joomla! Extension
- Model extraction

11

SESSION THE EDITOR

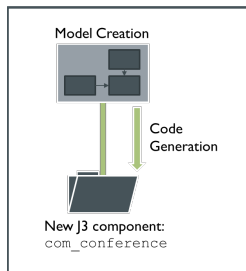


DENNIS PRIEFER, WOLFGANG ROST 09.12.17

- Alternative IDE plugins
 - Eclipse
 - PhpStorm
 - IntelliJ IDEA
- Model extraction has to be executed manually with jext2jml
- <https://github.com/icampus/jooMDD>

12

SESSION USE CASE I – DEVELOPMENT OF A JOOMLA COMPONENT



DENNIS PRIEFER, WOLFGANG ROST 09.12.17

- Data Modelling (with references)
- Page Modelling
- Extension Modelling
- Code Generation
- Installation to Joomla 3.8 Site
- Create Data

- Open the web editor: tinyurl.com/joomdd-web
 - Getting Started
 - Create an account (recommended)

13

SESSION USE CASE I – DEVELOPMENT OF A JOOMLA COMPONENT

- Data Modelling (with references)

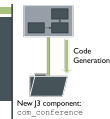
Entities require a unique attribute

```
Entity Participant {
  attributes {
    Attribute name {
      type = Text
      Unique attribute with ID
    }
    Attribute affiliation {
      type = Text
    }
    Attribute nationality {
      type = Text
    }
    Attribute address {
      type = Text
    }
  }
}
```

```
Entity Tail {
  attributes {
    Attribute title {
      type = Text
      Unique attribute
    }
    Attribute description {
      type = Text
    }
    Attribute speaker {
      type = Text
    }
  }
  references {
    Reference {
      EntityAttribute = speaker
      Entity = Participant
      EntityReference = Participant.name
      lower = 1
      upper = 1
    }
  }
}
```

Reference to another entity

^ must be used, if a name is a keyword. It will not be generated.



DENNIS PRIEFER, WOLFGANG ROST 09.12.17

14

Figure G.4: Hands-on Tutorial Presentation (4)

SESSION

USE CASE I – DEVELOPMENT OF A JOOMLA COMPONENT

Code Generation

New J! component: com_conference

Page Modelling

Entity reference

View table columns

Link to edit page

```

IndexPage Participants {
  *Entity Participant
  table columns = Participant.name,
    Participant.address,
    Participant.affiliation
  filters = Participant.name, Participant.affiliation
  links
  InternalContentLink Details {
    target = Participant linked attribute = Participant.name
    linkparameters {
      Parameter name = *Attribute *Participant.name*
    }
  }
}

DetailPage Participant {
  *Entity Participant
  links (...)
}

IndexPage Talks {
  *Entity Talk
  table columns = Talk.title, Talk.speaker
  links (...)
}

DetailPage Talk {
  *Entity Talk
  links (...)
}

IndexPage Rooms (...)
DetailPage Room (...)

```

ID	Status	Name	Address	Affiliation
1	<input checked="" type="checkbox"/>	John Doe	Gießen	THM
2	<input checked="" type="checkbox"/>	Zurbitgen Anderson	US	Harvard University
3	<input checked="" type="checkbox"/>	Zalfr Green	USA	Stanford University
4	<input checked="" type="checkbox"/>	Yalton Robinson	USA	Massachusetts Institute of Technology

DENNIS PRIEFER, WOLFF ROST 09.12.17

15

SESSION

USE CASE I – DEVELOPMENT OF A JOOMLA COMPONENT

Code Generation

New J! component: com_conference

Extension Modelling

Create your individual component

```

Component MyConference_Dennis {
  Manifestation {
    authors {
      Author "John Doe" {
        authoremail = "john.doe@example.org"
      }
    }
    copyright = "Copyright (C) 2017 All right reserved."
    license = "GNU General Public license"
    version = "1.0.1"
  }
  languages {
    Language de-DE {
    }
    Language en-GB {
    }
  }
}

```

```

sections {
  Frontend section {
    *Pages {
      *Page : Participants
      *Page : Talks
      *Page : Rooms
      *Page : Programme
    }
  }
  Backend section {
    *Pages {
      *Page : Participant
      *Page : Talks
      *Page : Talk
      *Page : Rooms
      *Page : Room
      *Page : Programme
      *Page : Session
    }
  }
}

```

Component views

DENNIS PRIEFER, WOLFF ROST 09.12.17

16

SESSION

USE CASE I – DEVELOPMENT OF A JOOMLA COMPONENT

Code Generation

New J! component: com_conference

Save Model

Generate Code

Load Example Model

Generate Joomla! 3.x code for the selected model.

```

main.eJSL
1 eJSLModel "Conference" {
2   eJSL part: CMS Extension {
3     entities {
4       Entity Participant {
5         attributes {
6           Attribute name {
7             type = Short_Text
8             Unique attribute with ID
9           }
10          Attribute affiliation {

```

My Workspace

reverse

src

main.eJSL

src-gen

Extensions

Conference

com_myconference_denni

DENNIS PRIEFER, WOLFF ROST 09.12.17

17

SESSION

USE CASE I – DEVELOPMENT OF A JOOMLA COMPONENT

Code Generation

New J! component: com_conference

Reinstallation to Joomla 3.8 Site

<http://icampus.thm.de/j3/administrator>

User: jdev

PW: jdevmeeting2017

Create data for your own component

ID	Status	Name	Address	Affiliation
1	<input checked="" type="checkbox"/>	John Doe	Gießen	THM
2	<input checked="" type="checkbox"/>	Zurbitgen Anderson	US	Harvard University
3	<input checked="" type="checkbox"/>	Zalfr Green	USA	Stanford University
4	<input checked="" type="checkbox"/>	Yalton Robinson	USA	Massachusetts Institute of Technology
5	<input checked="" type="checkbox"/>	Yancy Roberts	USA	University of California, Berkeley
6	<input checked="" type="checkbox"/>	Yadrow Green	London	University of Cambridge
7	<input checked="" type="checkbox"/>	Gall Banks	5509 Lakeview St	University of Chicago
8	<input checked="" type="checkbox"/>	Carte Stephens	8221 Hebron Pkwy	Yale University
9	<input checked="" type="checkbox"/>	Greg King	Switzerland	Swiss Federal Institute of Technology Zurich

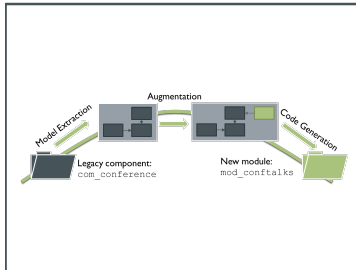
DENNIS PRIEFER, WOLFF ROST 09.12.17

18

Figure G.5: Hands-on Tutorial Presentation (5)

265

SESSION USE CASE 2 – AUGMENTATION OF AN EXISTING COMPONENT BY A NEW MODULE

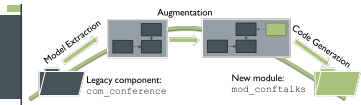


DENNIS PRIEFER, WOLFF ROST 09.12.17

24

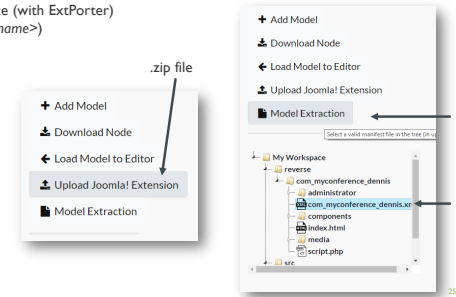
- Export component from Joomla instance
- Model extraction with JooMDD editor
- Augmentation of the model
- Generation of new module
- Installation to Joomla 3.8 site
- Configure the module to be shown in the frontend

SESSION USE CASE 2 – AUGMENTATION OF AN EXISTING COMPONENT BY A NEW MODULE



- Export component from Joomla instance (with ExtPorter)
(legacy extension: *MyConference_<your name>*)
→ Use your generated component

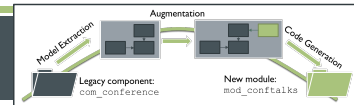
- Model extraction with JooMDD editor
 - Upload extension package
 - Select manifest file
 - Extract model



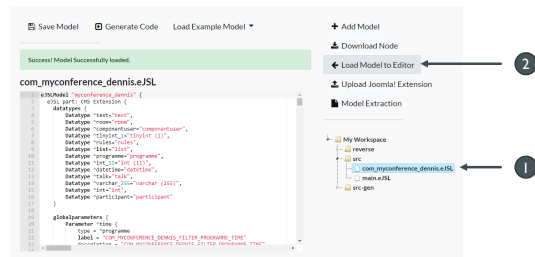
DENNIS PRIEFER, WOLFF ROST 09.12.17

25

SESSION USE CASE 2 – AUGMENTATION OF AN EXISTING COMPONENT BY A NEW MODULE



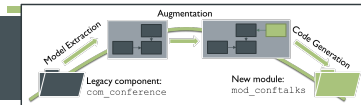
- Load extracted model to the editor



DENNIS PRIEFER, WOLFF ROST 09.12.17

26

SESSION USE CASE 2 – AUGMENTATION OF AN EXISTING COMPONENT BY A NEW MODULE



- Augmentation of the model (*confalks_<your name>*)



DENNIS PRIEFER, WOLFF ROST 09.12.17

27

Figure G.6: Hands-on Tutorial Presentation (6)

SESSION

USE CASE 2 – AUGMENTATION OF AN EXISTING COMPONENT BY A NEW MODULE

```

graph LR
    A[Legacy component: com_conference] -- Model Extraction --> B[ ]
    subgraph Augmentation
        B[ ] --> C[ ]
    end
    C -- Code Generation --> D[New module: mod_conftalks]
    style B fill:none,stroke:none
    style C fill:none,stroke:none
    
```



28

SESSION
USE CASE 2 – AUGMENTATION OF AN EXISTING COMPONENT BY A NEW MODULE



- Configure your module to be placed in the frontend



2

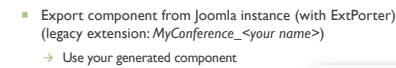
SESSION
USE CASE 3 – MIGRATION OF AN EXISTING COMPONENT FROM JOOMLA 3 TO JOOMLA 4



- 30

SESSION
USE CASE 3 – MIGRATION OF AN EXISTING COMPONENT FROM JOOMLA 3 TO JOOMLA 4

The diagram illustrates the migration process from Joomla 3 to Joomla 4. It shows a flow from a 'Legacy J3 component: com_FooBar-1.0' to a 'New J4 component: com_FooBar-1.0'. The process involves 'Model Extraction' and 'Refactorings (optional)', leading to 'Code Generation'.



- **Alternative:** use your extracted model from use case 2



10

267



Optional Refactorings

```

387 }
388
389 extensions {
390   Component "myconference_dennis_J4" {
391     Manifestation {
392       authors {
393         Author "Dennis Priefer" {
394           authoremail = "dennis.priefer@example.org"
395           authorurl = ""
396         }
397       }
398       copyright = "Copyright (C) 2017 All right reserved."
399       license = "GNU General Public License"
400     }
401     languages {
402       Language de-DE {}
403       Language en-GB {}
404     }
405     sections {
406       Frontend section {
407         *Pages {
408           *Page : "participants"
409           *Page : "programme"
410         }
411       }
412     }
413   }
414 }
  
```

DENNIS PRIEFER, WOLF ROST 09.12.17



Generation of new J4 extension

Save Model Generate Code Joomla 3 Joomla 4 Load Example Model

Success! Model Successfully loaded.

com_myconference_dennis.eJSL

```

1 eJSLModel "myconference_dennis" {
2   eJSL part: CMS Extension {
3     datatypes {
4       Datatype "text"="text",
5       Datatype "rooms"="room",
6       Datatype "componentusers"="componentuser",
7       Datatype "tinyint_1"="tinyint (1)",
8       Datatype "rules"="rules",
9       Datatype "list"="list",
10    }
11  }
12 }
  
```

DENNIS PRIEFER, WOLF ROST 09.12.17



Have fun with Joomla 4 ☺

- Controller
- Field
- forms
- Helper
- language
- Model
- sql
- Table
- tmpl
- View
- access
- config
- dispatcher

```

<?php
namespace Joomla\Component\myconference\Administrator\View\talk;
/** @version 1.0.1 ... */

defined( name: 'JTEXTC' ) or die('Restricted access');

use Joomla\CMS\MVC\View\HtmlView as BaseHtmlView;
use Joomla\CMS\Language\Text;

/**
 * @description talkView for myconference
 */
import('joomla.filesystem.path');
/** myconferenceView class for component com_myconference ... */
class HtmlView extends BaseHtmlView
{
  }
  
```

DENNIS PRIEFER, WOLF ROST 09.12.17

THANK YOU

TOOLS: [GITHUB.COM/ICAMPUS/JOOMDD](https://github.com/ICampus/JOOMDD)
[TINYURL.COM/JOOMDD-WEB](https://tinyurl.com/joomdd-web)

Dennis Priefer
dennis.priefer@mni.thm.de

Wolf Rost
wolf.rost@mni.thm.de

THM
TECHNISCHE HOCHSCHULE MITTELHESSEN

Figure G.8: Hands-on Tutorial Presentation (8)

Bibliography

- [1] N. Adermann and J. Boggiano. Composer - Introduction, 2019. URL: <https://getcomposer.org/doc/00-intro.md>.
- [2] AIIM. What is Enterprise Content Management (ECM)?, 2021. URL: <https://www.aiim.org/resources/glossary/enterprise-content-management>.
- [3] Akeeba Ltd. Akeeba Backup - Homepage, 2020. URL: <https://www.akeebabackup.com/>.
- [4] M. Alnasser. JCCreator, 2019. URL: <https://jc-creator.com>.
- [5] R. Andersen and T. Batova. The Current State of Component Content Management: An Integrative Literature Review. *IEEE Transactions on Professional Communication*, 58(3):247–270, 2015.
- [6] M. Antkiewicz, T. T. Bartolomei, and K. Czarnecki. Automatic extraction of framework-specific models from framework-based application code. In K. Stirewalt, A. Egyed, and B. Fischer, editors, *Proceedings of the twenty-second IEEEACM international conference on Automated software engineering*, page 214, New York, NY, 2007. ACM.
- [7] M. Antkiewicz and K. Czarnecki. Framework-Specific Modeling Languages with Round-Trip Engineering. In O. Nierstrasz, editor, *Model driven engineering languages and systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 692–706, Berlin, 2006. Springer.
- [8] M. Antkiewicz, K. Czarnecki, and M. Stephan. Engineering of Framework-Specific Modeling Languages. *IEEE Transactions on Software Engineering*, 35(6):795–824, 2009.
- [9] J. Antrim. thm-mni-ii/com_organizer, 2020. URL: https://github.com/thm-mni-ii/com_organizer.
- [10] AOE. Open Source CMS Vergleich: TYPO3 vs. Drupal vs. Joomla! vs. WordPress, 2019. URL: <https://www.aoe.com/de/knowledge-base/cms-vergleich-typo3-vs-drupal-vs-joomla-vs-wordpress.html>.
- [11] Apache Friends. Bitnami for XAMPP, 2019. URL: https://www.apachefriends.org/bitnami_for_xampp.html.
- [12] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *Model driven engineering languages and systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135, Berlin and Heidelberg, 2010. Springer.
- [13] A. Avram. Gartner’s Software Hype Cycles for 2012, 2012. URL: <https://www.infoq.com/news/2012/08/Gartner-Hype-Cycle-2012>.
- [14] T. Baar. The Definition of Transitive Closure with OCL – Limitations and Applications –. In M. Broy and A. V. Zamulin, editors, *Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 358–365, Berlin and Heidelberg, 2003. Springer.
- [15] H. Bagheri and K. Sullivan. Bottom-up model-driven development. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1221–1224, Piscataway, NJ, 2013. IEEE.
- [16] P. Baker, S. Loh, and F. Weil. Model-Driven Engineering in a Large Industrial Context — Motorola Case Study. In L. Briand, editor, *Model driven engineering languages and*

- systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 476–491, Berlin and Heidelberg, 2005. Springer.
- [17] D. Barker. *Web content management: Systems, features, and best practices*. O'Reilly, Beijing and Boston, first edition edition, 2016.
 - [18] V. R. Basili, R. W. Selby, and D. H. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE-12(7):733–743, 1986.
 - [19] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance*, pages 368–377, 1998.
 - [20] S. Bechtold, S. Brannen, J. Link, M. Merdes, M. Philipp, J. de Rancourt, and C. Stein. JUnit 5 User Guide, 2020. URL: <https://junit.org/junit5/docs/current/user-guide/>.
 - [21] K. Beck, M. Beedle, A. van Bennekum, W. Cockburn, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for Agile Software Development, 2001. URL: <http://agilemanifesto.org/>.
 - [22] S. Bergmann. PHPLOC, 2020. URL: <https://github.com/sebastianbergmann/phploc>.
 - [23] M. L. Bernardi, G. A. D. Lucca, and D. Distant. Model-driven fast prototyping of RIAs: From conceptual models to running applications. In *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 250–258, 2014.
 - [24] L. Bettini. *Implementing domain-specific languages with Xtext and Xtend: Learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices*. Packt Publ., Birmingham, 2013.
 - [25] B. Bigendako and E. Syriani. Modeling a Tool for Conducting Systematic Reviews Iteratively. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*, volume 1, pages 552–559, Funchal, 2018. Scitepress.
 - [26] B. W. Boehm. Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, 1(1):75–88, 1984.
 - [27] M. Brambilla. *Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann, Waltham, MA, 2015.
 - [28] M. Brambilla, J. Cabot, and M. Wimmer. *Model-driven software engineering in practice*. Morgan & Claypool Publishers, San Rafael, California, second edition edition, 2017.
 - [29] H. Brunelière, J. Cabot, G. Dupé, and F. Madiot. MoDisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012–1032, 2014.
 - [30] H. Brunelière, J. Cabot, F. Jouault, and F. Madiot. MoDisco: a generic and extensible framework for model driven reverse engineering. In C. Pecheur, editor, *Proceedings of the IEEEACM international conference on Automated software engineering*, pages 173–174, New York, NY, 2010. ACM.
 - [31] C. Bunse, H. Gross, and C. Peper. Embedded System Construction – Evaluation of Model-Driven and Component-Based Development Approaches. In M. R. V. Chaudron, editor, *Models in Software Engineering*, pages 66–77, Berlin, Heidelberg, 2009. Springer.
 - [32] H. Burden, R. Heldal, and J. Whittle. Comparing and Contrasting Model-driven Engineering at Three Large Companies. In *Proceedings of the 8th ACM/IEEE International*

- Symposium on Empirical Software Engineering and Measurement, ESEM '14*, pages 14:1–14:10, New York, NY, USA, 2014. ACM.
- [33] D. Buytaert. Drupal Homepage, 2018. URL: <https://www.drupal.org/>.
 - [34] D. Buytaert. Module project index | Drupal.org, 2020.
 - [35] J. Cabot. What is the exact difference between Meta Model and DSL in context of Model driven software development?, 2017. URL: <https://www.quora.com/What-is-the-exact-difference-between-Meta-Model-and-DSL-in-context-of-Model-driven-software-development>.
 - [36] J. Cabot. jcabot/WordPress-Plugin-DSL, 2019. URL: <https://github.com/jcabot/WordPress-Plugin-DSL>.
 - [37] J. Cabot. WordPress Plugin DSL and code-generator, 2019. URL: <https://seriouswp.com/wordpress-plugin-dsl-and-code-generator/>.
 - [38] J. Canovas. Domain-Specific Languages, 2013. URL: <https://fr.slideshare.net/zirrus/domainspecific-languages>.
 - [39] J. Canovas. jlcanovas/gra2mol, 2014. URL: <https://github.com/jlcanovas/gra2mol>.
 - [40] J. L. Cánovas Izquierdo and J. García Molina. Extracting models from source code in software modernization. *Software & Systems Modeling*, 13(2):713–734, 2014.
 - [41] S. Ceri. *Designing data-intensive Web applications*. Morgan Kaufmann Publishers, Amsterdam and Boston, 2010.
 - [42] S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks*, 33(1-6):137–157, 2000.
 - [43] G. Charness, U. Gneezy, and M. A. Kuhn. Experimental methods: Between-subject and within-subject design. *Economic Behavior & Organization*, 81(1):1–8, 2012.
 - [44] A. Chauhan. Joomla 3 and Joomla 4 features comparison: 15 new things to take note | JoomlaArt, 2017. URL: <https://www.joomlaart.com/blog/joomla-3-vs-joomla-4-features-comparison-15-new-things-to-take-note>.
 - [45] G. Chénard, I. Khriss, and A. Salah. Towards the automatic discovery of platform transformation templates of legacy object-oriented systems. In *Proceedings of the 6th International Workshop on Models and Evolution*, pages 51–56, New York, 2012. ACM.
 - [46] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *12th International IEEE Enterprise Distributed Object Computing Conference, 2008*, pages 222–231, Piscataway, NJ, 2008. IEEE.
 - [47] T. Clark and J. Warmer. *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*. Springer Berlin Heidelberg, 2003.
 - [48] P. Clements and L. Northrop. *Software product lines: Practices and patterns*. Addison-Wesley, Boston and San Francisco and New York and Toronto and Montreal and London and Munich and Paris and Madrid and Capetown and Sydney and Tokyo and Singapore and Mexico City, 7. print edition, 2009.
 - [49] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Taylor and Francis, Hoboken, 2nd ed. edition, 2013.
 - [50] G. W. Corder and D. I. Foreman. *Nonparametric Statistics: A Step-by-Step Approach*. Wiley, 2nd ed. edition, 2014.

- [51] D. Crockford. JSLint: The JavaScript Code Quality Tool, 2018. URL: <https://www.jshint.com/>.
- [52] K. Czarnecki and U. W. Eisenecker. *Generative programming: Methods, tools, and applications*. Addison-Wesley, Boston Mass. u.a., 6. print edition, 2005.
- [53] M. D. Del Fabro, J. Bézivin, F. Jouault, and P. Valduriez. Applying Generic Model Management to Data Mapping. In *BDA*, 2005.
- [54] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-oriented reengineering patterns*. Square Bracket Associates, Switzerland, revised [ed.] edition, 2009.
- [55] F. J. Domínguez-Mayo, M. J. Escalona, M. Mejías, M. Ross, and G. Staples. Quality evaluation for Model-Driven Web Engineering methodologies. *Information and Software Technology*, 54(11):1265–1282, 2012.
- [56] Drupal community. Drupal: A "Hello World" Custom Page Module, 2016. URL: <https://www.drupal.org/docs/8/creating-custom-modules/a-hello-world-custom-page-module>.
- [57] Drupal community. Drupal: Agile Unit Testing, 2019. URL: <https://www.drupal.org/docs/8/phpunit/agile-unit-testing>.
- [58] G. Dupe and H. Bruneliere. Eclipse MoDisco, 2019. URL: <https://www.eclipse.org/MoDisco/>.
- [59] S. Efftinge and M. Spoenemann. Xtend - Modernized Java, 2015. URL: <http://www.eclipse.org/xtend/>.
- [60] S. Efftinge and M. Spoenemann. Xtext - Language Engineering Made Easy!, 2016. URL: <https://eclipse.org/Xtext/>.
- [61] eJ Technologies. Java Profiler - JProfiler, 2020. URL: <https://www.ej-technologies.com/products/jprofiler/overview.html>.
- [62] S. Evers, J. Ernsting, and T. A. Majchrzak. Towards a Reference Architecture for Model-Driven Business Apps. In T. X. Bui and R. H. Sprague, editors, *Proceedings of the 49th Annual Hawaii International Conference on System Sciences*, pages 5731–5740, Piscataway, NJ, 2016. IEEE.
- [63] Facebook Inc. React – A JavaScript library for building user interfaces, 2019. URL: <https://reactjs.org/>.
- [64] M. C. Feathers. *Working effectively with legacy code*. Prentice Hall Professional Technical Reference, Upper Saddle River, NJ, 10. print edition, 2009.
- [65] A. Fernandez, S. Abrahão, and E. Insfran. Empirical Validation of a Usability Inspection Method for Model-driven Web Development. *Systems and Software*, 86(1):161–186, 2013.
- [66] P. Filipe, A. Ribeiro, and A. R. da Silva. XIS-CMS: Towards a model-driven approach for developing platform-independent CMS-specific modules. In *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 535–543, 2016.
- [67] J. Fons, V. Pelechano, M. Albert, and Ó. Pastor. Development of Web Applications from Web Enhanced Conceptual Schemas. In I. Song, S. W. Liddle, T. Ling, and P. Scheuermann, editors, *Conceptual Modeling - ER 2003*, volume 2813 of *Lecture Notes in Computer Science*, pages 232–245, Berlin and Heidelberg, 2003. Springer.
- [68] OpenJS Foundation. Node.js, 2019. URL: <https://nodejs.org/en/>.

- [69] The Eclipse Foundation. ATL, 2018. URL: <http://www.eclipse.org/at1/>.
- [70] M. Fowler and K. Beck. *Refactoring: Improving the design of existing code*. Addison-Wesley, Boston, 28. printing edition, 2013.
- [71] M. Fowler and R. Parsons. *Domain-specific languages*. Addison-Wesley, Upper Saddle River, NJ and Boston and Indianapolis and San Francisco and New York and Toronto and Montreal and London and Munich and Paris and Madrid and Sydney and Tokyo and Singapore and Mexico City, 2011.
- [72] R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In L. C. Briand and A. L. Wolf, editors, *Future of software engineering, 2007*, pages 37–54, Los Alamitos, Calif., 2007. IEEE Computer Society.
- [73] D. S. Frankel. *Model driven architecture: Applying MDA to enterprise computing*. Wiley, Indianapolis, Ind., 2003.
- [74] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 1995.
- [75] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing Model Adaptation by Precise Detection of Metamodel Changes. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA '09*, pages 34–49, Berlin, Heidelberg, 2009. Springer.
- [76] Gartner Inc. Hype Cycle Research Methodology, 2017. URL: <http://www.gartner.com/technology/research/methodologies/hype-cycle.jsp>.
- [77] M. Geirhos. *Entwurfsmuster: Das umfassende Handbuch*. Rheinwerk Verlag, Bonn, 2nd ed. edition, 2018.
- [78] GitHub. GitHub Actions, 2020. URL: <https://github.com/features/actions>.
- [79] Google. Angular, 22.02.2019. URL: <https://angular.io/>.
- [80] Google Ireland Limited. Blogger website, 2019. URL: <https://www.blogger.com>.
- [81] T. Greifenberg, K. Müller, A. Roth, B. Rumpe, C. Schulze, and A. Wortmann. Modeling variability in template-based code generators for product line engineering. In A. Oberweis and R. Reussner, editors, *Modellierung 2016*, pages 141–156, Bonn, 2016. Gesellschaft für Informatik e.V.
- [82] M. Grieger. *Model-driven software modernization: concept-based engineering of situation-specific methods*. PhD thesis, University of Paderborn, Germany, 2016.
- [83] R. Gronback. Eclipse Modeling Project, 2017. URL: <https://www.eclipse.org/modeling/emf/>.
- [84] V. Gruhn, D. Pieper, and C. Röttgers. *MDA: Effektives Software-Engineering mit UML2 und Eclipse*. Springer Berlin Heidelberg, 2006.
- [85] L. Haoyi. lihaoyi/fastparse, 2020. URL: <https://github.com/lihaoyi/fastparse>.
- [86] A. S. Hawley. scala/scala-xml, 2020. URL: <https://github.com/scala/scala-xml>.
- [87] P. Hegedus. Revealing the effect of coding practices on software maintainability. In *2013 IEEE International Conference on Software Maintenance*, pages 578–581, 2013.
- [88] J. Helming and M. Koegel. EMFStore, 01.11.2017. URL: <http://www.eclipse.org/emfstore/>.

- [89] B. Henderson-Sellers, J. Ralyté, P. J. Agerfalk, and M. Rossi. *Situational method engineering*. Springer Berlin Heidelberg, 2014.
- [90] M. Herrmannsdoerfer. Eclipse Edapt, 2018. URL: <https://www.eclipse.org/edapt/>.
- [91] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In B. Malloy, S. Staab, and M. van den Brand, editors, *Software Language Engineering*, pages 163–182, Berlin, Heidelberg, 2011. Springer.
- [92] T. P. Hettmansperger and J. W. McKean. *Robust Nonparametric Statistical Methods*. CRC Press, 2010.
- [93] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In D. Lo, S. Apel, and S. Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 426–437, Piscataway, NJ, 2016. IEEE.
- [94] J. Hossler, M. Born, and S. Saito. Significant Productivity Enhancement through Model Driven Techniques: A Success Story. In *10th IEEE International Enterprise Distributed Object Computing Conference, 2006*, pages 367–373, Los Alamitos, Calif., 2006. IEEE Computer Society.
- [95] J. Hutchinson, M. Rouncefield, and J. Whittle. Model-driven engineering practices in industry. In R. N. Taylor, H. Gall, and N. Medvidović, editors, *33rd International Conference on Software Engineering (ICSE)*, pages 633–642, Piscataway, NJ, 2011. IEEE.
- [96] IBM. Rational Rhapsody Developer, 2017. URL: <http://www-03.ibm.com/software/products/de/ratirhap>.
- [97] International Organization for Standardization. ISO/IEC 25000:2005(en), Software Engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE, 2005. URL: <https://www.iso.org/obp/ui/#iso:std:iso-iec:25000:ed-1:v1:en>.
- [98] International Organization for Standardization. ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, 2011.
- [99] International Organization for Standardization. ISO/IEC 25000:2014 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE, 2014. URL: <https://www.iso.org/standard/64764.html>.
- [100] iTeam. Joomla 4 tutorials: Orthogonal Component Structure Revolution, 2020. URL: <https://www.jomsocial.com/blog/joomla-4-tutorials-orthogonal-component-structure-revolution>.
- [101] J. J. Jacoby. SLASH Architecture – My approach to building WordPress plugins, 2012. URL: <https://jjj.blog/2012/12/slash-architecture-my-approach-to-building-wordpress-plugins/>.
- [102] Jensen Technologies SL. Component Creator, 2019. URL: <https://www.component-creator.com>.
- [103] JetBrains. IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains, 2018. URL: <https://www.jetbrains.com/idea/>.
- [104] JetBrains. PhpStorm: Lightning-Smart IDE for PHP Programming by JetBrains, 2018. URL: <https://www.jetbrains.com/phpstorm/>.

- [105] Joomla! community. Joomla! Documentation, 2019. URL: https://docs.joomla.org/Main_Page.
- [106] Joomla! community. Joomla! Framework - Github Project, 2019. URL: <https://github.com/joomla-framework>.
- [107] Joomla! community. Manifest files – Joomla! Documentation, 2019. URL: https://docs.joomla.org/Manifest_files.
- [108] Joomla! community. Part 2: The Administrator code – Joomla! Documentation, 2019. URL: https://docs.joomla.org/Part_2:_The_Administrator_code.
- [109] Joomla! community. J3.x:Developing an MVC Component/Developing a Basic Component – Joomla! Documentation, 2020. URL: https://docs.joomla.org/J3.x:Developing_a_n_MVC_Component/Developing_a_Basic_Component.
- [110] Joomla! community. Joomla 4 Alpha is Out, 2020. URL: <https://magazine.joomla.org/issues/issue-nov-2017/item/3290-joomla-4-alpha-is-out>.
- [111] Joomla! community. Joomla CodeSniffer – Joomla! Documentation, 2020. URL: https://docs.joomla.org/Joomla_CodeSniffer.
- [112] Joomla! community. Model-View-Controller – Joomla! Documentation, 2020. URL: <https://docs.joomla.org/Model-View-Controller>.
- [113] Joomla! community. Plugin – Joomla! Documentation, 2020. URL: <https://docs.joomla.org/Plugin>.
- [114] S. Jörges. *Construction and evolution of code generators: A model-driven and service-oriented approach*. Springer Berlin Heidelberg, 2013.
- [115] B. Joshi, P. Budhathoki, W. L. Woon, and D. Svetinovic. Software Clone Detection Using Clustering Approach. In S. Arik, T. Huang, W. K. Lai, and Q. Liu, editors, *Neural information processing*, volume 9490 of *Lecture Notes in Computer Science*, pages 520–527, Cham and Heidelberg and New York and Dordrecht and London, 2015. Springer.
- [116] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.
- [117] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like Transformation Language. In P. Tarr, editor, *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720, New York, NY, 2006. ACM.
- [118] N. Juristo and A. M. Moreno. *Basics of Software Engineering Experimentation*. Springer Publishing Company, Incorporated, 2010.
- [119] A. Kaczmarek. How to use custom fields in Joomla 3 ?, 2019. URL: <https://www.joomla-monster.com/documentation/joomla-tutorials/custom-fields-in-joomla-3-7>.
- [120] N. Kahani. MDE TOOLS, 2017. URL: <http://www.mdetools.com/>.
- [121] N. Kahani, M. Bagherzadeh, J. Dingel, and J. R. Cordy. The Problems with Eclipse Modeling Tools: A Topic Analysis of Eclipse Forums. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS ’16, pages 227–237, New York, NY, USA, 2016. ACM.
- [122] Nafiseh Kahani, Mojtaba Bagherzadeh, James R. Cordy, Juergen Dingel, and Daniel Varró. Survey and classification of model transformation tools. *Software & System Modeling*, 18(4):2361–2397, 2019.

- [123] Nafiseh Kahani and James R. Cordy. Comparison and Evaluation of Model Transformation Tools. Technical report, Queen's University, Kingston, Ontario, 2015.
- [124] C. Kapser. Toward an Understanding of Software Code Cloning as a Development Practice, 2009. URL: <http://hdl.handle.net/10012/4753>.
- [125] C. Kapser and M. Godfrey. "Cloning Considered Harmful" Considered Harmful. In H. Selvaraj, editor, *19th International Conference on Systems Engineering*, pages 19–28, Piscataway, NJ, 2008. IEEE.
- [126] S. Karg, A. Raschke, M. Tichy, and G. Liebel. Model-driven software engineering in the openETCS project. In B. Baudry and B. Combemale, editors, *19th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 238–248, New York, New York, 2016. The Association for Computing Machinery.
- [127] G. Karsai, H. Krah, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel. Design Guidelines for Domain Specific Languages. In M. Rossi, J. Sprinkle, J. Gray, and J.-P. Tolvanen, editors, *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM '09)*, Helsinki, 2014. Helsingin Kauppakorkeakoulu.
- [128] T. Katsimpa, Y. Panagis, E. Sakkopoulos, G. Tzimas, and A. Tsakalidis. Application modeling using reverse engineering techniques. In H. M. Haddad, editor, *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1250–1255, New York, NY, 2006. ACM.
- [129] T. Kehrer. SiLift: Semantic Lifting of Model Differences, 2017. URL: <http://pi.informatik.uni-siegen.de/Projekte/SiLift/>.
- [130] T. Kehrer, U. Kelter, and G. Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In P. Alexander, editor, *26th IEEE/ACM International Conference on Automated Software Engineering*, pages 163–172, Piscataway, NJ, 2011. IEEE.
- [131] S. Kelly, K. Lyytinen, and M. Rossi. MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, editors, *Advanced Information Systems Engineering*, pages 1–21, Berlin, Heidelberg, 1996. Springer.
- [132] S. Kelly and J. Tolvanen. *Domain-specific modeling: Enabling full code generation*. Wiley IEEE and IEEE Xplore, Hoboken, New Jersey and Piscataway, New Jersey, 2008.
- [133] A. Kempkens. Joomla! - Content Management System to build websites & apps, 2019. URL: <https://www.joomla.org/about-joomla.html>.
- [134] S. Khatoon, G. Li, and A. Mahmood. Comparison and evaluation of source code mining tools and techniques: A qualitative approach. *Intelligent Data Analysis*, 17(3):459–484, 2013.
- [135] A. G. Kleppe. *Software language engineering: Creating domain-specific languages using metamodels*. Addison-Wesley, Upper Saddle River, NJ, 2009.
- [136] G. Kotonya and I. Sommerville. *Requirements engineering: Processes and techniques*. Wiley, Chichester u.a., 2004.
- [137] A. Kraus, A. Knapp, and N. Koch. *Model-Driven Generation of Web Applications in UWE*. PhD thesis, Ludwig-Maximilians-Universität, Munich, Germany, 2008.
- [138] C. Krause. Henshin Project Page, 2020. URL: <https://www.eclipse.org/henshin/>.

- [139] K. Krogmann and S. Becker. A Case Study on Model-Driven and Conventional Software Development: The Palladio Editor. In S. Böttinger, L. Theuvsen, S. Rank, and M. Morgenstern, editors, *Software Engineering 2007 – Beiträge zu den Workshops – Fachtagung des GI-Fachbereichs Softwaretechnik*, pages 169–175, Bonn, 2007. Gesellschaft für Informatik e. V.
- [140] V. Kulkarni, S. Barat, and U. Ramteerthkar. Early Experience with Agile Methodology in a Model-Driven Approach. In J. Whittle, T. Clark, and T. Kühne, editors, *Model Driven Engineering Languages and Systems*, pages 578–590, Berlin, Heidelberg, 2011. Springer.
- [141] LF Projects, LLC. Home - Zend Framework, 2019. URL: <https://getlaminas.org/>.
- [142] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson. Model-based Engineering in the Embedded Systems Domain: An Industrial Survey on the State-of-practice. *Software & Systems Modeling*, 17(1):91–113, 2018.
- [143] K. Linne and S. Schepp. ejsl-angular-interpreter, 2020. URL: <https://git.thm.de/klenn67/ejsl-angular-interpreter/wikis/home>.
- [144] Magento. Magento - eCommerce Software & eCommerce Platform Solutions[online]. Available: <https://magento.com>, 2018. URL: <https://magento.com/>.
- [145] Mancini, J. The next wave: Moving from ECM to Intelligent Information Management, 2017. URL: <http://info.aiim.org/the-next-wave-from-ecm-to-intelligent-information-management-aiim>.
- [146] R. C. Martin, M. C. Feathers, T. R. Ottinger, J. J. Langr, B. L. Schuchert, J. W. Grenning, and K. D. Wampler. *Clean code: A handbook of agile software craftsmanship*. Prentice Hall, Upper Saddle River, NJ and Boston and Indianapolis and San Francisco and New York and Toronto and Montreal and London and Munich and Paris and Madrid, Capetown and Sydney and Tokyo and Singapore and Mexico City, 2009.
- [147] Y. Martinez, C. Cachero, and S. Meliá. Empirical Study on the Maintainability of Web Applications: Model-driven Engineering vs Code-centric. *Empirical Software Engineering*, 19(6):1887–1920, 2014.
- [148] J. Martinez-Caro, A. Aledo-Hernandez, A. Guillen-Perez, R. Sanchez-Iborra, and M. Cano. A Comparative Study of Web Content Management Systems. *Information*, 9(2):27, 2018.
- [149] MathWorks. MathWorks - Makers of MATLAB and Simulink, 2017. URL: <https://uk.mathworks.com/>.
- [150] S. McKeever. Understanding Web content management systems: Evolution, lifecycle and market. *Industrial Management & Data Systems*, 103(9):686–692, 2003.
- [151] M. Meike, J. Sametinger, and A. Wiesauer. Security in Open Source Web Content Management Systems. *IEEE Security & Privacy Magazine*, 7(4):44–51, 2009.
- [152] S. Mendel, A. Knowles, B. Cook, and E. Enke. SQL_Parser, 2016. URL: https://pear.php.net/package/SQL_Parser.
- [153] T. Mens and P. van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.
- [154] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.
- [155] MicroTool. objectiF – Model-driven Development, 2017. URL: <https://www.microtool.de/en/objectif-model-driven-development/>.

- [156] P. Mohagheghi, W. Gilani, A. Stefanescu, and M. A. Fernandez. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. *Empirical Software Engineering*, 18(1):89–116, 2013.
- [157] N. Mohd Razali and B. Yap. Power Comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling Tests. *Statistical Modeling and Analytics*, 2(1):21–33, 2011.
- [158] D. C. Montgomery. *Design and analysis of experiments*. John Wiley & Sons, New York / Chichester, 5th edition, 2001.
- [159] Netcraft Ltd. Web Server Survey, 2019. URL: <https://news.netcraft.com/archives/category/web-server-survey/>.
- [160] M. C. Norrie, L. Di Geronimo, A. Murolo, and M. Nebeling. The Forgotten Many? A Survey of Modern Web Development Practices. In S. Casteleyn, G. Rossi, and M. Winckler, editors, *Current Trends in Web Engineering*, volume 8541 of *Lecture Notes in Computer Science*, pages 290–307, Cham, 2014. Springer International Publishing.
- [161] Object Management Group. About the Abstract Syntax Tree Metamodel Specification Version 1.0, 2011. URL: <https://www.omg.org/spec/ASTM/>.
- [162] Object Management Group. OMG, Object Management Group, 2017. URL: <http://www.omg.org/>.
- [163] Object Management Group. About the Knowledge Discovery Metamodel Specification Version, 2018. URL: <https://www.omg.org/spec/KDM/1.0/>.
- [164] Object Management Group. IFML: The Interaction Flow Modeling Language | The OMG standard for front-end design, 2018. URL: <https://www.ifml.org/>.
- [165] Object Management Group. Meta Object Facility Specification Version 2.5.1, 2019. URL: <https://www.omg.org/spec/MOF/>.
- [166] Inc Open Source Matters. Joomla Coding Standards Documentation, 2018. URL: <https://developer.joomla.org/coding-standards.html>.
- [167] Inc Open Source Matters. Joomla! Framework, a framework for developing PHP applications, 2019. URL: <https://framework.joomla.org/>.
- [168] Inc. Open Source Matters. J3.x:Developing an MVC Component/Developing a Basic Component – Joomla! Documentation, 2020. URL: https://docs.joomla.org/J3.x:Developing_an_MVC_Component/Developing_a_Basic_Component.
- [169] Inc Open Source Matters. J4 Component example - Mywalks – Joomla! Documentation, 2020. URL: https://docs.joomla.org/J4_Component_example_-_Mywalks.
- [170] Inc. Open Source Matters. Joomla! Extensions Directory, 2020. URL: <https://extensions.joomla.org/>.
- [171] Inc Open Source Matters. Joomla!.org, ToDo. URL: <https://www.joomla.org/>.
- [172] Open Source Matters Inc. Joomla 4 is on the horizon Idots Alpha 12, 2019. URL: <https://developer.joomla.org/news/793-joomla-4-is-on-the-horizon-alpha-12.html>.
- [173] M. Otto, J. Thornton, and Bootstrap contributors. Bootstrap, 2020. URL: <https://getbootstrap.com/>.
- [174] T. Otwell. Laravel - The PHP Framework For Web Artisans, 2019. URL: <https://laravel.com/>.

- [175] F. Paetsch, A. Eberlein, and F. Maurer. Requirements engineering and agile software development. In *Twelfth IEEE International Workshops on Enabling Technologies*, pages 308–313, Los Alamitos, Calif, 2003. IEEE Computer Society.
- [176] J. I. Panach, S. España, Ó. Dieste, Ó. Pastor, and N. Juristo. In search of evidence for model-driven development claims: An experiment on quality, effort, productivity and satisfaction. *Information and Software Technology*, 62:164–186, 2015.
- [177] P. E. Papotti, A. F. do Prado, W. L. de Souza, C. E. Cirilo, and L. F. Pires. A Quantitative Analysis of Model-Driven Code Generation through Software Experimentation. In C. Salinesi, M. C. Norrie, and Ó. Pastor, editors, *Advanced Information Systems Engineering*, pages 321–337, Berlin, Heidelberg, 2013. Springer.
- [178] S. Patig and J. Dibbern. Requirements Engineering — Enzyklopaedie der Wirtschaftsinformatik, 2018. URL: <http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Hauptaktivitaten-der-Systementwicklung/Problemanalyse-/Requirements-Engineering/index.html>.
- [179] PHP Framework Interop Group. PSR-12: Extended Coding Style, 2020. URL: <https://www.php-fig.org/psr/psr-12/>.
- [180] M. Pichler. PHP Mess Detector, 2016. URL: <https://phpmd.org/>.
- [181] Joomla! PLT. Weblinks - Joomla Extension Directory, 2015. URL: <https://extensions.joomla.org/extension/weblinks/>.
- [182] T. Preston-Werner. Semantic Versioning 2.0.0, 2019. URL: <https://semver.org/>.
- [183] D. Priefer. Model-driven development of content management systems based on Joomla. In I. Crnkovic, M. Chechik, and P. Grünbacher, editors, *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 911–914, New York, 2014. ACM.
- [184] D. Priefer. JooMDD project site (GitHub), 2019. URL: <https://github.com/thm-mni-ii/JooMDD>.
- [185] D. Priefer, P. Kneisel, W. Rost, D. Struber, and G. Taentzer. Applying MDD in the Content Management System Domain: Scenarios and Empirical Assessment. In *ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems*, pages 56–66. IEEE, 2019.
- [186] D. Priefer, P. Kneisel, and D. Strüber. Iterative Model-Driven Development of Software Extensions for Web Content Management Systems. In A. Anjorin and H. Espinoza, editors, *Modelling Foundations and Applications: 13th European Conference, ECMFA 2017, Held as Part of STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings*, pages 142–157, Cham, 2017. Springer International Publishing.
- [187] D. Priefer, P. Kneisel, and G. Taentzer. A Model-Driven Process to Migrate Web Content Management System Extensions. In A. Bozzon, editor, *Web engineering*, volume 9671 of *Lecture Notes in Computer Science Information systems and applications, incl. Internet/Web, and HCI*, pages 603–606, Cham and Heidelberg, 2016. Springer.
- [188] D. Priefer, P. Kneisel, and G. Taentzer. JooMDD: A Model-Driven Development Environment for Web Content Management System Extensions. In *ICSE Companion '16: Companion Proceedings of the 38th International Conference on Software Engineering*, pages 633–636, New York, NY, USA, 2016. ACM.
- [189] D. Priefer, W. Rost, D. Strüber, G. Taentzer, and P. Kneisel. Online Appendix: Model-Driven Development in the Content Management System Domain: Empirical Assessment

- during Common Development Scenarios, 2020. URL: https://figshare.com/articles/journal_contribution/Model-Driven_Development_in_the_Content_Management_System_Domain_Empirical_Assessment_during_Common_Development_Scenarios/12661538.
- [190] D. Priefer, W. Rost, D. Strüber, G. Taentzer, and P. Kneisel. Applying MDD in the content management system domain: Scenarios, tooling, and a mixed-method empirical assessment. *Software & Systems Modeling*, 2021.
- [191] V. Proietti and MooTools Developers. MooTools, 2020. URL: <https://mootools.net/>.
- [192] S. Robertson and J. Robertson. *Mastering the requirements process: Getting requirements right*. Addison-Wesley, Upper Saddle River, N.J, 3rd ed. edition, 2013.
- [193] A. Rodrigues da Silva, J. Saraiva, R. Silva, and C. Martins. XIS-UML Profile for eXtreme Modeling Interactive Systems. In J. Fernandes, editor, *Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 55–66, Los Alamitos, Calif., 2007. IEEE Computer Society.
- [194] R. Rodriguez-Echeverria, J. Preciado, J. Sierra, J. M. Conejero Manzano, and F. Sánchez-Figueroa. AutoCRUD: Automatic generation of CRUD specifications in interaction flow modelling language. *Science of Computer Programming*, 168:165–168, 2018.
- [195] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Model Migration with Epsilon Flock. In *Proceedings of the Third International Conference on Theory and Practice of Model Transformations*, ICMT’10, pages 184–198, Berlin, Heidelberg, 2010. Springer.
- [196] W. Rost. Mining of DSLs and Generator Templates from Reference Applications. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS ’20, pages 1–7, New York, NY, USA, 2020. Association for Computing Machinery.
- [197] A. Roth and B. Rumpe. Towards product lining model-driven development code generators. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 539–545, 2015.
- [198] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.
- [199] P. Rubens. Use Low-Code Platforms to Develop the Apps Customers Want, 2014. URL: <https://www.cio.com/article/2845378/development-tools/use-low-code-platforms-to-develop-the-apps-customers-want.html>.
- [200] C. Rupp. *Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis*. Hanser, München and Wien, 5., aktualisierte und erw. Aufl. edition, 2009.
- [201] I. Salman, A. T. Misirli, and N. Juristo. Are students representatives of professionals in software engineering experiments? In *IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 666–676, 2015.
- [202] J. Sánchez Cuadrado, O. Ávila García, J. L. Cánovas Izquierdo, and A. Sánchez-Barbudo Herrera. On the automation of the horse-shoe model for software modernization, 2012. URL: <https://modeling-languages.com/on-the-automation-of-the-horse-shoe-model-for-software-modernization/>.

- [203] Ó. Sánchez Ramón, J. Sánchez Cuadrado, and J. García Molina. Model-driven reverse engineering of legacy graphical user interfaces. *Automated Software Engineering*, 21(2):147–186, 2014.
- [204] J. Saraiva and A. R. da Silva. CMS-Based Web-Application Development Using Model-Driven Languages. In *Proceedings of the 2009 Fourth International Conference on Software Engineering Advances*, ICSEA '09, pages 21–26, Washington DC USA, 2009. IEEE Computer Society.
- [205] J. Saraiva and A. R. da Silva. Web-Application Modeling With the CMS-ML Language. In *II Simpósio de Informática (INForum 2010)*, 2010.
- [206] J. d. S. Saraiva. *Development of CMS-based Web Applications with a Multi-Language Model-Driven Approach*. PhD thesis, Universidade Técnica de Lisboa, Lisbon, Portugal, 2012.
- [207] Scala Community. scala/scala-parser-combinators, 2020. URL: <https://github.com/scala/scala-parser-combinators>.
- [208] A. Schauerhuber, M. Wimmer, E. Kapsammer, W. Schwinger, and W. Retschitzegger. Bridging WebML to model-driven engineering: From document type definitions to meta object facility. *IET Software*, 1(3):81–97, 2007.
- [209] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing legacy systems: Software technologies, engineering processes, and business practices*. Addison-Wesley, Boston, MA, 2003.
- [210] SensioLabs. Symfony, High Performance PHP Framework for Web Development, 2019. URL: <https://symfony.com/>.
- [211] S. S. Shapiro and M. B. Wilk. An Analysis of Variance Test for Normality. *Biometrika*, 52(3-4):591–611, 1965.
- [212] Shopify International Limited. Shopify Homepage, 2019. URL: <https://www.shopify.com/>.
- [213] F. Shull, D. I. K. Sjøberg, and J. Singer. *Guide to Advanced Empirical Software Engineering*. Springer-Verlag London Limited, London, 2008.
- [214] Simply Open Source. Component Architect, 2019. URL: <https://www.componentarchitect.com>.
- [215] K. Smolander, K. Lyytinen, V. Tahvanainen, and P. Marttiin. MetaEdit— A flexible graphical environment for methodology modelling. In R. Andersen, J. A. Bubenko, and A. Sølvsberg, editors, *Advanced Information Systems Engineering*, pages 168–193, Berlin, Heidelberg, 1991. Springer.
- [216] V. Sousa, E. Syriani, and M. Paquin. Feedback on How MDE Tools Are Used Prior to Academic Collaboration. In *Proceedings of the Symposium on Applied Computing*, SAC '17, pages 1190–1197, New York, NY, USA, 2017. ACM.
- [217] Spacedog ApS. Component Generator, 2019. URL: <https://www.componentgenerator.com/>.
- [218] SparxSystems Software GmbH. Enterprise Architect Editionen, 2016. URL: <https://www.sparxsystems.de/uml/enterprise-architect-editionen/>.
- [219] Squarespace Ireland Ltd. Squarespace website, 2019. URL: <https://de.squarespace.com/>.
- [220] Squiz Labs. PHP CodeSniffer, 2021. URL: https://github.com/squizlabs/PHP_CodeSniffer.

- [221] T. Stahl and M. Völter. *Model-driven software development: Technology, engineering, management*. John Wiley, Chichester, England and Hoboken, NJ, 2006.
- [222] C. Stoermer, F. Bachmann, and C. Verhoef. SACAM: The Software Architecture Comparison Analysis Method. URL: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6605>.
- [223] Strapi. Strapi - Open source Node.js Headless CMS, 2020. URL: <https://strapi.io/>.
- [224] J. Svajlenko and C. K. Roy. CloneWorks: A Fast and Flexible Large-Scale Near-Miss Clone Detection Tool. In *IEEE/ACM 39th International Conference on Software Engineering companion*, pages 177–179, Piscataway, NJ, 2017. IEEE.
- [225] V. Svansson and R. E. Lopez-Herrejon. A Web Specific Language for Content Management Systems. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, Montréal, Canada, 2007.
- [226] J. Swanhart and A. Rothe. greenlion/PHP-SQL-Parser, 2019. URL: <https://github.com/greenlion/PHP-SQL-Parser>.
- [227] The Eclipse Foundation. EMF Compare - Compare and Merge Your EMF Models, 10.11.2017. URL: <https://www.eclipse.org/emf/compare/>.
- [228] The Eclipse Foundation. Epsilon Flock, 2014. URL: <http://www.eclipse.org/epsilon/doc/flock/>.
- [229] The Eclipse Foundation. EuGENia, 2014. URL: <https://www.eclipse.org/epsilon/doc/eugenia/>.
- [230] The Eclipse Foundation. Sirius - The easiest way to get your own Modeling Tool, 2018. URL: <https://www.eclipse.org/sirius/>.
- [231] THM Web Editorial Staff. THM Organizer - Home, 2020. URL: <https://www.thm.de/organizer/>.
- [232] H. C. Thode. *Testing For Normality*. Taylor & Francis, 2002.
- [233] M. Torchiano, F. Tomassetti, F. Ricca, A. Tiso, and G. Reggio. Benefits from modelling and MDD adoption. In *Proceedings of the Second Edition of the International Workshop on Experiences and Empirical Studies in Software Modelling*, pages 1–6, New York, NY, 2012. ACM.
- [234] J. Torres, J. Resendiz, I. Aedo, and J. M. Doderio. ORIGINAL ARTICLE: A Model-Driven Development Approach for Learning Design Using the LPCEL Editor. *Journal of King Saud University - Computer and Information Sciences*, 26(1):17–27, 2014.
- [235] F. Trias. Building CMS-based Web applications using a model-driven approach. In C. Roland, J. Castro, and Ó. Pastor, editors, *Sixth International Conference on Research Challenges in Information Science*, pages 1–6, Piscataway, NJ, 2012. IEEE.
- [236] F. Trias, V. de Castro, M. López-Sanz, and E. Marcos. A Systematic Literature Review on CMS-based Web Applications. In *ICSOF*, 2013.
- [237] F. Trias, V. de Castro, M. Lopez-Sanz, and E. Marcos. Migrating Traditional Web Applications to CMS-based Web Applications. *Electronic Notes in Theoretical Computer Science*, 314:23–44, 2015.
- [238] F. Trias, V. de Castro, M. López-Sanz, and E. Marcos. RE-CMS: a reverse engineering toolkit for the migration to CMS-based web applications. In *SAC '15: Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 810–812, New York, NY, USA, 2015. ACM.

- [239] TYPO3 Association. TYPO3 - The Enterprise Open Source CMS [online]. Available: <https://typo3.org>, 2019. URL: <https://typo3.org/>.
- [240] A. Vargha and H. D. Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.
- [241] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. In E. Di Nitto, M. Harman, and P. Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816, New York, NY, 2015. ACM.
- [242] Vast Development Method. Joomla Component Builder, 2019. URL: <https://www.joomlacomponentbuilder.com/>.
- [243] S. Vaupel. *A Framework for Model-Driven Development of Mobile Applications with Context Support*. PhD thesis, Philipps-Universität, Marburg, 2018.
- [244] S. Vaupel, D. Strüber, F. Rieger, and G. Taentzer. Agile bottom-up development of domain-specific ides for model-driven development. In D. di Ruscio, J. de Lara, and A. Pierantonio, editors, *Proceedings of the Workshop on Flexible Model Driven Engineering co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages & Systems*, CEUR Workshop Proceedings, pages 12–21. CEUR-WS.org, 2015.
- [245] S. Vaupel, G. Taentzer, R. Gerlach, and M. Guckert. Model-driven development of mobile applications for Android and iOS supporting role-based app variability. *Software & System Modeling*, 17(1):35–63, 2018.
- [246] S. D. Vermolen, G. Wachsmuth, and E. Visser. Generating database migrations for evolving web applications. In E. Denney and U. P. Schultz, editors, *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, SIGPLAN notices, pages 83–92, New York, NY, 2011. ACM Press.
- [247] E. Visser. WebDSL: A Case Study in Domain-Specific Language Engineering. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and transformational techniques in software engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373, Berlin, Heidelberg, 2008. Springer.
- [248] K. Vlaanderen, F. Valverde, and O. Pastor. Model-Driven Web Engineering in the CMS Domain: A Preliminary Research Applying SME. In W. Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, C. Szyperski, J. Filipe, and J. Cordeiro, editors, *Enterprise Information Systems*, volume 19 of *Lecture Notes in Business Information Processing*, pages 226–237, Berlin, Heidelberg, 2009. Springer.
- [249] J. M. Vlissides. *Pattern hatching: Design patterns applied*. Addison-Wesley, Reading, Mass., 1998.
- [250] M. Voelter. Best Practices for DSLs and Model-Driven Development, 2009. URL: http://www.jot.fm/issues/issue_2009_09/column6/index.html.
- [251] W3Techs. Historical yearly trends in the usage of content management systems, January 2018, 2020. URL: https://w3techs.com/technologies/history_overview/content_management/all/y.
- [252] W3Techs. Usage Statistics and Market Share of Content Management Systems for Websites, January 2018, 2020. URL: https://w3techs.com/technologies/overview/content_management/all.

- [253] C. Wagner. *Model-Driven Software Migration: A Methodology: Reengineering, Recovery and Modernization of Legacy Systems*. Springer Fachmedien Wiesbaden, Wiesbaden and s.l., 2014.
- [254] J. B. Warmer and A. G. Kleppe. *The object constraint language: Precise modeling with UML*. Addison-Wesley, Boston, 1999.
- [255] T. Warneke. JSQlParser/JSqLParser, 2020. URL: <https://github.com/JSQlParser/JSqLParser>.
- [256] WebRatio. Webratio homepage. URL: <https://www.webratio.com/site/content/en/home>.
- [257] G. M. Weinberg. *Systemdenken und Softwarequalität*. Hanser, München and Wien, 1994.
- [258] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal. Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem? In A. Moreira, B. Schätz, J. Gray, and A. Vallecillo, editors, *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science / Programming and Software Engineering*, pages 1–17, Berlin/Heidelberg, 2013. Springer.
- [259] Wix.com Ltd. Wix Homepage, 2019. URL: <https://wix.com/>.
- [260] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.
- [261] WooCommerce. WooCommerce - eCommerce for WordPress, 2019. URL: <https://woocommerce.com/>.
- [262] WordPress community. Make WordPress - Get Involved, 2019. URL: <https://make.wordpress.org/>.
- [263] WordPress community. Plugin Developer Handbook, 2019. URL: <https://developer.wordpress.org/plugins/>.
- [264] WordPress.org. Installing WordPress - Famous 5 Minutes Installation, 2019. URL: https://codex.wordpress.org/Installing_WordPress#Famous_5-Minute_Installation.
- [265] WordPress.org. WordPress Versions, 2019. URL: https://codex.wordpress.org/WordPress_Versions.
- [266] WordPress.org. WordPress Plugins, 2020. URL: <https://de.wordpress.org/plugins/>.
- [267] E. You. Vue.js, 2019. URL: <https://vuejs.org/>.
- [268] G. Zhang, X. Peng, Z. Xing, and W. Zhao. Cloning practices: Why developers clone and what can be changed. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 285–294, Piscataway, NJ, 2012. IEEE.
- [269] Y. Zhang and S. Patel. Agile Model-Driven Development in Practice. *IEEE Software*, 28(2):84–91, 2011.
- [270] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu. The impact of continuous integration on other software development practices: A large-scale empirical study. In G. Rosu, M. Di Penta, and T. N. Nguyen, editors, *32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 60–71, Piscataway, NJ, 2017. IEEE.
- [271] M. F. Zibran and C. K. Roy. Towards flexible code clone detection, management, and refactoring in IDE. In J. R. Cordy, editor, *Proceedings of the 5th International Workshop on Software Clones*, pages 75–76, New York, NY, 2011. ACM.

